ISSN 0265-2919

90p

87

THE HOME COMPUTER ADVANCED COURSE

MAKING THE MOST OF YOUR MICRO

An ORBIS Publication

IR£1.15 Aus $2.15 NZ $2.65 SA R2.45 Sing $4.50

CONTENTS

APPLICATION

MODEL MAKING Expert-Ease allows you to develop expert systems on a home micro. We suggest some useful applications and put the program through its paces — 1721

HARDWARE

TO AND FRO Our series on computer components looks at input/output facilities — 1732

SOFTWARE

WORKING WITH WINDOWS To conclude our series on the GEM operating environment, we discuss some of the programming facilities it provides — 1726

SQUEEZED IN Despite memory limitations, many word processing programs for home micros are very impressive — 1735

COMPUTER SCIENCE

DIRECT CONTROL We begin a series on C, a programming language that bridges the gap between high and low-level languages — 1724

JARGON

TABLE TO TELECONFERENCING A weekly glossary of computing terms — 1728

PROGRAMMING PROJECTS

CARDS UP! We give the card display and shuffling routines for the BBC Micro, Spectrum and Amstrad machines — 1729

MACHINE CODE

CHANGING ADDRESS We complete our discussion of the range of addressing modes available to the 68000 programmer — 1738

Next Week

- MS-DOS has become the de facto standard operating system for business machines. We begin a series that takes an overview of the system.
- We take a look at the transmission of data and high resolution images via the cable network.
- In our 68000 machine code series, we take a more detailed look at aspects of the instruction set.

QUIZ

1) What function is performed by the GEM VDI (virtual device interface)?

2) Why are PIO chips incorporated into the processor's memory map?

3) Why is it advisable for an expert system to be able to explain its reasoning?

Answers To Last Week's Quiz

1) Step-time programming is a process whereby the user can 'write' a piece of music into a computer by typing in the notes and specifying the time signature.
2) Because WordStar is installed as a 'transient program' it is able to use many of the facilities provided by the operating system (CP/M), thus saving space and maintaining compatibility with other programs.
3) Because the 68000 has only a 24-bit address bus, only this figure is meaningful when writing assembly language programs.
4) Parsing — also known as syntax analysis — is the process by which the interpreter or computer examines a line of code to ensure it is legally structured.

Coming Up

- A review of the Commodore Amiga.
- A series on the Unix operating system.

FACT SHEET We give the first of three fact sheets that detail the instruction set used on the Motorola 68000 chip — INSIDE BACK COVER

Editor Stephen Cooke; **Art Editor** Claudia Zeff; **Deputy Editor** Steve Colwill; **Production Editor** Bobby Pickering; **Designer** Julian Dorr; **Staff Writer** Steve Malone; **Art Assistant** Caroline Clayton; **Sub Editor** Jon Kaye; **Contributors** Mike Curtis, Steve Colwill, Steve Malone, Nigel Cross, David Fensome, Surya, Peter Shaw; **Software Consultants** Pilot Software City; **Group Art Director** Perry Neville; **Managing Director** Stephen England; **Published by Orbis Publishing Ltd; Editorial Director** Brian Innes; **Project Development** Peter Brookesmith; **Executive Editor** Maurice Geller; **Production Assistant** Susan Brown; **Subscription Manager** Christine Allen; **Designed and produced by Bunch Partworks Ltd; Editorial Office** 14 Rathbone Place, London W1P 1DE; **© APSIF Copenhagen 1985; © Orbis Publishing Ltd 1985: Typeset by Universe; Reproduction by Mullis Morgan Ltd; Printed in Great Britain by Heanor Gate Printing Ltd, Derby**

HOW TO OBTAIN ISSUES AND BINDERS FOR THE HOME COMPUTER ADVANCED COURSE – Issues can be obtained by placing an order with your newsagent or direct from our subscription department. If you have any difficulty obtaining any back issues from your newsagent, please write to us stating the issue(s) required and enclosing a cheque for the cover price of the issue(s). **AUSTRALIA** – please write to: Gordon & Gotch (Aus) Ltd, 114 William Street, PO Box 767G, Melbourne, Victoria 3001. **MALTA, NEW ZEALAND & SOUTH AFRICA –** Back numbers are available at cover price from your newsagent. In case of difficulty, write to the address given for binders.
UK/EIRE – Issue Price: 90p/IR£1.15. Subscription: 6 months: £26.00. 1 Year: £52.00. Binder: please send £3.95 per binder, or take advantage of our special offer in early issues. **EUROPE** – Issue Price: 90p. Subscription: 6 months air: £44.72. Surface: £36.14. 1 year air: £89.44. Surface: £72.28. Binder: £5.00. Airmail: £8.25. **MALTA** – Obtain binders from your newsagent or Miller (Malta) Ltd, MA Vassalli Street, Valetta, Malta. Price: £3.95. **MIDDLE EAST** – Issue Price: 90p. Subscription: 6 months air: £50.18. Surface: £36.14. 1 year air: £100.36. Surface: £72.28. Binder: £5.00. Airmail: £8.25. **AMERICAS/ASIA/AFRICA** – Issue Price: US/CAN$1.95/90p. Subscription: 6 months air: £59.54. Surface: £36.14. 1 year air: £119.08. Surface: £72.28. Binder: £5.00. Airmail: £9.50. **SOUTH AFRICA** – Issue Price: SA R2.45. Obtain binders from any branch of Central News Agency or Intermag, PO Box 57394, Springfield 2137. **SINGAPORE** – Issue Price: Sing $4.50. Obtain binders from MPH Distributors, 601 Sims Drive, 03-07-21, Singapore 1438. **AUSTRALASIA/FAR EAST** – Issue Price: 90p. Subscription: 6 months air: £64.22. Surface: £36.14. 1 year air £128.44. Surface: £72.28. Binder: £5.00. Airmail: £9.75. **AUSTRALIA** – Issue Price: Aus$2.15. Obtain binders from First Post Pty Ltd, 23 Chandos Street, St Leonards, NSW 2065. **NEW ZEALAND** – Issue Price: NZ$2.65. Obtain binders from your newsagent or Gordon & Gotch (NZ) Ltd, PO Box 1595, Wellington.
ADDRESS FOR BINDERS AND BACK ISSUES – Orbis Publishing Limited, Orbis House, Bedfordbury, London WC2 4BT. Telephone 01-379 5211. Cheques/postal orders should be made payable to Orbis Publishing Limited. Binder prices include postage and packing and prices are in stirling. Back issues are sold at the cover price, and we do not charge carriage in the UK.
NOTE – Binders and back issues are obtainable subject to availability of stocks. Whilst every attempt is made to keep the price of the issues and binders constant, the publishers reserve the right to increase the stated prices at any time when circumstances dictate. Binders depicted in this publication are those produced for the UK and Australian markets only. Binders and Issues may be subject to import duty and/or local taxes, which are not included in the above prices unless stated.
ADDRESS FOR SUBSCRIPTIONS – Orbis Publishing Limited, Hurst Farm, Baydon Road, Lambourn Woodlands, Newbury Berks, RG16 7TW. Telephone: 0488-72666. All cheques/postal orders should be made payable to Orbis Publishing Limited. Postage and packaging is included in subscription rates, and prices are given in sterling.

COVER PHOTOGRAPHY BY MARCUS WILSON-SMITH

MODEL MAKING

As we saw in our introduction to the subject on page 1301, expert systems have a wide variety of applications, but are also exceedingly expensive to produce. We look here at a product that enables novice users to tailor and interrogate an expert system for applications of their own.

Expert systems tend, by definition, to be applied in particular fields, such as medical diagnosis or geological surveyance. But the expert systems generator offers the user the chance to apply this 'soft' technology to more immediate problems that might not otherwise warrant the expense of setting up a customised system. A good example of this approach is provided by Expert-Ease, from Thorn-EMI Software, which allows you to create your own expert system (known as a model) and run it on an IBM PC or ACT Sirius, with a minimum of 128 Kbytes of RAM.

As we showed on page 1301, the section of an expert system that does all the work is the inference engine — the logic processing module. It's the programming of this section that would normally place the expert system approach out of reach of most users, unless the expense for programming the intended application was deemed absolutely imperative. But Expert-Ease actually creates an inference engine for you, so that all the hard work is done by the computer rather than the person creating the model.

There are four main approaches to designing the inference engine of an expert system: rule-based formalism, using an IF. . .THEN structure; semantic nets, based on sets and subsets ('flat battery' for example, is a member of the set 'electrical failure'); frames, which are set up like a traditional database except that specified values in specified fields cause the system to draw conclusions about the record; and horn clauses, the predicate logic on which PROLOG is based (see page 1272). Expert-Ease uses a fifth, automated, method — the Analogue Concept Learning System (ACLS).

The ACLS was developed as a stand-alone module at the artificial intelligence laboratory of Edinburgh University by staff and consultants of Intelligent Terminals Ltd, a company owned by Professor Donald Mitchie, who is generally acknowledged as one of the world's leading authorities on artificial intelligence. A discussion of the workings of the ACLS is beyond the scope of this piece, but for those who are interested it was developed from Iterative Dichotomiser 3 (ID3), a public-domain program written in PASCAL.

Why are the leaves on my plant yellow and drooping ?
a. Has it got greenfly
b. Too much water ?
c. Is it getting enough light ?
XXXXXXXXX

CHRIS CORR

A SIMPLE PROBLEM

Expert-Ease models are really created backwards, in so far as you must first decide what all the possible conclusions are, then decide what questions need to be answered by the user, and finally decide which answers will lead to which conclusions. As an example of Expert-Ease's structure, let's look at a very simple application — solving the problem of whether or not we should go to the football match today. Despite its obvious limitations, tackling this problem demonstrates the potential of the system for applications in the more demanding areas of diagnosis and decision-making.

The two possible conclusions to this problem are obviously 'go' and 'don't go', so the next thing to do is decide what questions we need to ask or, to put it another way, what factors are important in making the decision. Again, keeping it simple, let's assume that we're only interested in good weather and having enough time. Let's begin by drawing a decision table:

Example/ Question	Good weather?	Enough time?	Decision
1.	Y	Y	GO
2.	Y	N	NO
3.	N	Y	NO
4.	N	N	NO

Expert Guidance
Expert-Ease, an expert systems generator from Thorn-EMI, enables expert systems to be quickly and inexpensively tailored for different applications. Particularly useful in diagnostics and problem analysis, the power of the expert system — although previously confined to areas with a high financial profile — can now be applied in more humble situations

We can see that if the answer to either question is no, the decision is no. We can alter our table thus:

Example/ Question	Good weather?	Enough time?	Decision
1.	Y	Y	GO
2.	Y	N	NO
3.	N	*	NO
4.	N	*	NO

where the asterisk means 'don't care'. So if the answer to the first question is no, the second answer is irrelevant and needn't be asked.

This is, in fact, exactly how Expert-Ease requires us to enter information. First, we create a skeleton table containing the questions and the possible outcomes (known as the 'attribute screen'), and then we fill this screen with example answers (known as the 'example screen'). The only difference is that Expert-Ease doesn't limit us to yes or no answers; we can have as many responses as we like. Questions are designed in a multiple-choice format, and we fill in the numbers in the example screen. Once we've created our example screen, Expert-Ease uses the ACLS to induce the logic behind the decision table to create a 'rule'. This rule is applied when the model is run.

The ACLS (known in the package as the 'induction routine') checks our decision table for conflicts and points these out to us (such as: 11 node rule formed, examples 1 5 6 contradict rule).

CHAINED MODELS

Expert-Ease is undoubtedly an extremely powerful and sophisticated program. For most common applications, the models it creates stand up to any dedicated expert system. Furthermore,

Creating A Model

The first stage in creating an Expert-Ease model is to design the 'skeleton' on which the decision chart will be based. Expert-Ease calls this the 'attribute screen' (called by pressing A from the filer screen). To create this screen, we have to decide which questions we need to ask and in what order. We give names to each of these questions and they are displayed as column headings.

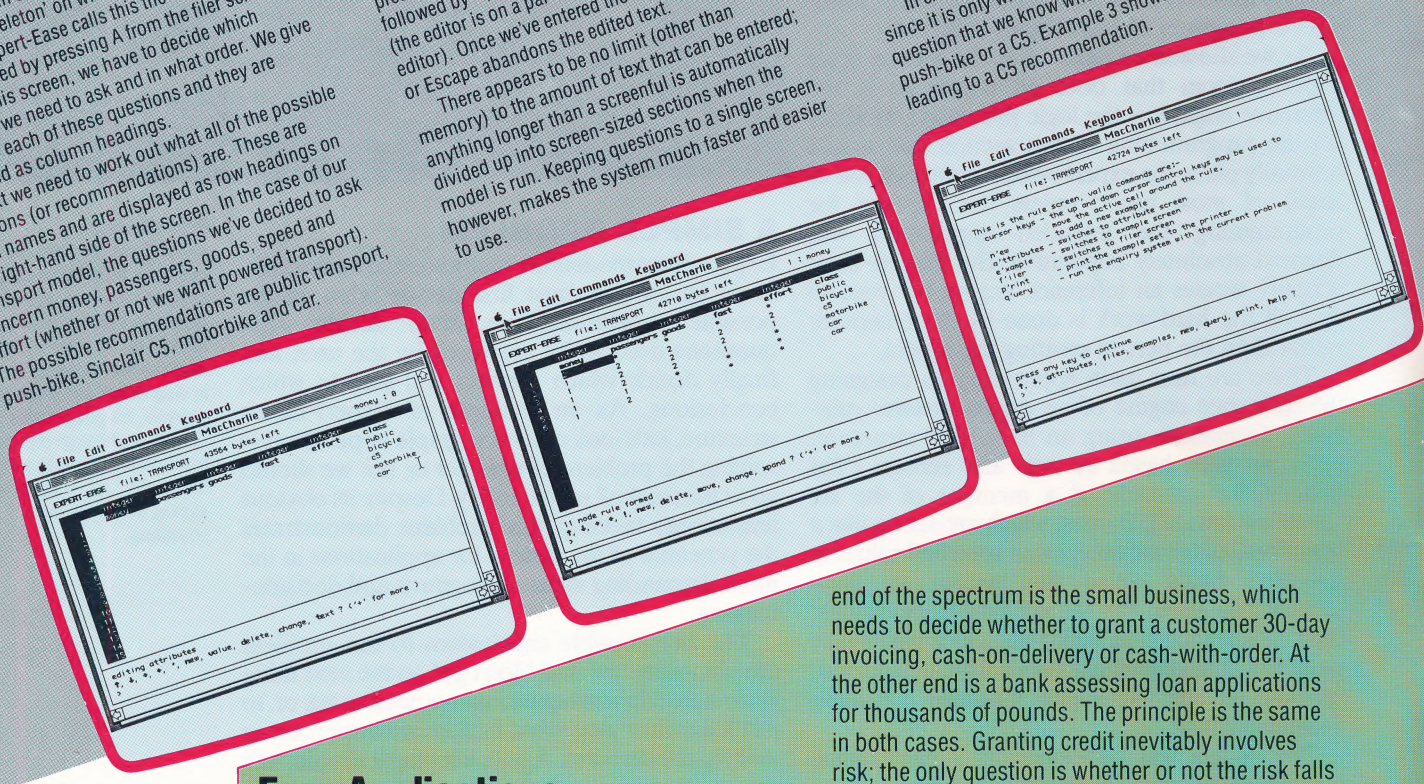
Next we need to work out what all of the possible decisions (or recommendations) are. These are given names and are displayed as row headings on the right-hand side of the screen. In the case of our transport model, the questions we've decided to ask concern money, passengers, goods, speed and effort (whether or not we want powered transport). The possible recommendations are public transport, push-bike, Sinclair C5, motorbike and car.

Once we've created these headings, we can enter the text for the questions and recommendations by pressing T (to display the accompanying text), followed by ' (a single apostrophe) to create or edit (the editor is on a par with the Commodore screen editor). Once we've entered the text, CTRL-C stores it or Escape abandons the edited text.

There appears to be no limit (other than memory) to the amount of text that can be entered; anything longer than a screenful is automatically divided up into screen-sized sections when the model is run. Keeping questions to a single screen, however, makes the system much faster and easier to use.

The example screen is the actual decision chart. It's so called because you show Expert-Ease by example how decisions are to be made. Thus, example 1 is a user answering 2 (no) to the money question, indicating that there is no capital to spare. Since the answers to the other questions are irrelevant, we place an asterisk (the 'don't care' symbol) under each of the other columns. Expert-Ease will then skip any question with an asterisk under it.

In example 2, we need to ask every question since it is only when we have the answer to the final question that we know whether the user requires a push-bike or a C5. Example 3 shows the answers leading to a C5 recommendation.



Easy Applications

The areas where Expert-Ease really comes into its own are in creating fairly simple models, particularly those which may have to be modified to keep up with future development.

Credit Applications

One area where expert systems can be extremely useful is in processing credit applications. At one end of the spectrum is the small business, which needs to decide whether to grant a customer 30-day invoicing, cash-on-delivery or cash-with-order. At the other end is a bank assessing loan applications for thousands of pounds. The principle is the same in both cases. Granting credit inevitably involves risk; the only question is whether or not the risk falls within acceptable limits.

The job done by the credit controller or loan officer is initially to assess the risk, and secondly to decide whether this risk falls within the limits acceptable to the company or bank. Creating an Expert-Ease model to perform the same function is simply a matter of categorising the criteria used to assess 'credit worthiness', devising questions for each, and using the answers to these questions to

models can be chained together, so that complex problems can be broken down into manageable pieces. The main difference between an Expert-Ease model and a custom-written expert system is that the former does not explain its own reasoning. You can, of course, create your own explanations in the text of the conclusion, but the user is not presented with the actual decision path of the model. This is not a disadvantage in systems designed to be used by non-experts, since the reasoning could be extremely complex, but it is significant in models designed to aid experts who might wish to challenge part or all of the reasoning.

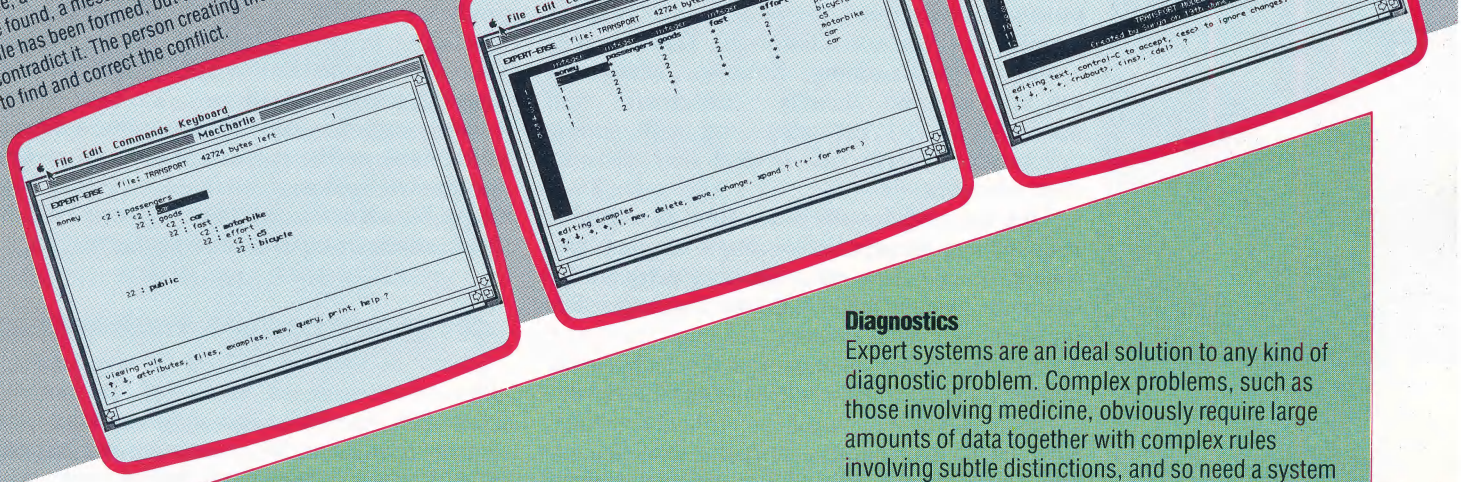
Once we've completed our examples, we enter an exclamation mark (!) to instruct Expert-Ease to use these examples to 'induce' the logic behind them. This has to be done before the model can be used. The induction process follows through each example looking for conflicts in logic (two or more paths leading to the same conclusion). Once this has been done, a 'rule' (flowchart) is formed. If any conflicts are found, a message is displayed to the effect that a rule has been formed, but examples x, y and z contradict it. The person creating the model then has to find and correct the conflict.

Once a rule has been formed (with or without contradictions), it can be displayed by pressing R for rule. It can be seen from this that the first question is stored under the heading 'money' and that a value of less than 2 (yes) leads to the 'passenger' question, while a value greater than or equal to 2 leads to the recommendation stored under 'public'. Viewing this screen is usually the easiest way to spot the problem if a rule has been formed with contradictions.

You can call up brief descriptions of Expert-Ease commands at any time by pressing H or ?. The exception to this is when you are halfway through filling in an example in the example screen; then it is necessary to complete the example first.

You will be asked which command you would like help with. Pressing H gives you a one-line summary of all the commands which are valid within the current screen. Pressing the first letter of any other command gives you a slightly longer description

SCREEN DUMPS FROM THE MACINTOSH WITH MACCHARLIE



assess the risk. So, for example, a simplified bank loan model might look something like this:

Security	Bad Debt	Overdrawn	Reg Income	Result
Y	*	*	*	OK
N	Y	*	*	NO
N	N	Y	Y	REFER
N	N	Y	N	NO
N	N	N	N	REFER
N	N	N	Y	OK

Thus, a typical run might look like this:

Can the customer offer security? No
Has the customer any history of bad debt? No
Has the customer ever had an unauthorised overdraft? Yes
Does the customer have a regular/verifiable income? Yes
Please refer this application to a loan officer.

In this way, straightforward applications can be handled by a junior member of staff using the expert system, while those requiring more detailed consideration are referred to a loan officer or manager. Should the bank ever change its loan criteria, such as requiring security on all loans over £1,000, the model can be quickly modified.

Diagnostics

Expert systems are an ideal solution to any kind of diagnostic problem. Complex problems, such as those involving medicine, obviously require large amounts of data together with complex rules involving subtle distinctions, and so need a system written from scratch. Less complex tasks, however, can be easily handled by Expert-Ease.

For example, a helicopter company flying oil rig workers to and from work uses Expert-Ease to assist in carrying out routine servicing of the helicopter fleet. Normally, when a helicopter is brought in for a minor routine service, a senior engineer is required to inspect the aircraft and determine the type of service required. Once this has been done, less senior mechanics can carry out the work. Using Expert-Ease, however, a mechanic can determine the work that needs to be carried out without consulting the senior engineer. The engineer merely makes his normal inspection once the work has been carried out; any mistake will be spotted then.

This same principle can be easily extended to the servicing and repair of all types of mechanical, electrical and electronic equipment — including, of course, computers. Indeed, Expert-Ease can be used to diagnose software faults as well as hardware problems. One use that springs immediately to mind in this area is in computer companies. Normally, when a customer telephones with a problem, trained hardware and software engineers have to spend valuable time diagnosing the fault. With Expert-Ease, however, this task could be left to junior staff, referring only the complex problems to the engineers

DIRECT CONTROL

There are two major types of programming language — high level on the one hand, such as BASIC and PASCAL, and low level on the other, such as assembly language, which directly controls the hardware. We begin a series on C, a programming language that bridges the gap between the two types.

Much of the development in high-level languages recently has been directed towards the creation of languages that better reflect human thought processes and remove the programmer as far as possible from the actual hardware. There remains a lot of programming, however, where the programmer is directly concerned with manipulating the hardware, and with getting maximum efficiency out of small and relatively slow machines.

This has traditionally been done in assembly language, or on occasions using a language like FORTRAN on hardware that has been designed for optimum performance with the language. The problem with using assembly language, however, is that although it gives maximum efficiency (in the hands of a good programmer), it is specific to one machine. There has long been a need, therefore, for a simple language at a fairly low-level that can be made easily available on a variety of machines, combining the structures and convenience of a high-level language with direct and efficient control over the hardware.

MARCUS WILSON-SMITH

C For Cheap
Hisoft-C runs on the Amstrad and Spectrum computers and is excellent value for money at around £35. The implementation is reasonably standard, with the serious exception that floating point numbers are not included. Hisoft is, however, planning a floating point version in the near future. The only other shortcoming is the manual, which although comprehensive is very poorly arranged and not suitable for beginners. Provided the user has prior experience or additional tutorial documentation, this is an excellent package

DEVELOPMENT OF C

The first language that could be said to fit these requirements was BCPL, developed by Martin Richards at Cambridge University. Many of the ideas from BCPL were used by Ken Thompson and others at Bell Labs in a language called B. At the time they were developing the Unix operating system, and needed a high-level language for this. B was developed into C by Kernighan and Ritchie, who published the definitive book *The C Programming Language* in 1978, which serves as a standard for C implementations — until a new international standard is agreed.

Unix was also developing and the language and the operating system have been intimately connected ever since. The 13,000 lines of code that make up the Unix kernel contain only about 800 lines of assembler, with C making up the rest. Another point in C's favour is that it is a relatively small language, which not only makes it easy to learn and use but also means that compilers are small and simple to write so that the language can be made available on a very wide variety of machines. Furthermore, since the language is already at a low level, the code produced is compact and fast. The result of this is that not only Unix and Unix applications are written in C but also the bulk of the MS-DOS and CP/M operating systems.

C compilers are available for nearly all micros as well as mainframes, and their use has enabled software houses to produce software that can be easily moved between such dissimilar machines as the Apple Macintosh and the IBM PC. Throughout this series we'll stick to the strict Kernighan and Ritchie version of the language and where appropriate, such as when we consider function libraries, assume the use of Unix — although most other versions of C follow this version as closely as possible.

A C program is built up by defining functions, each taking a number of arguments, values or pointers to values, which the function uses to produce a value or pointer that is returned. Each function is invoked simply by writing its name. The value returned can be used, although in some cases it may be discarded.

All C programs start by executing a function called main, so the outermost level of program must be a definition of this function. Here is a very simple but complete program to perform the first test on any new language or computer system:

```
main()
{
    printf("hello world\n");
}
```

Even this simple program brings out some important features of the language. The definition of a function consists of the function name followed by a list of its arguments in brackets. In this case, main happens not to have any arguments but the brackets must still be present.

The body of the function definition consists of a sequence of statements (only one in this case) enclosed in braces ({}), and each statement is terminated by a semicolon. The braces act in a similar fashion to Begin . . . End in PASCAL, in that they can be used to bracket any sequence of statements and declarations to form a compound statement, which in turn can be used in any position where a simple statement can be used. The semicolon is used in a different manner to PASCAL, but the rule is much simpler — each complete statement must be followed by a semicolon. The single statement in this program is a call to a library function, printf.

Input/output is not strictly defined in C because it varies so much from one machine to another. Instead, it's left to the particular implementations to provide what I/O they require in a library, though most follow the standard. The value returned by the printf function is of no use, so it's ignored. There can be any number of arguments for printf, whether numeric, character variables or strings (as in this case). Also, printf does not automatically issue a return, but C provides a

number of special symbols for control characters, all of which begin with a backslash (\). The return symbol (or line) is \n, as shown in the Escape Sequences table.

The following program would give exactly the same output as the first:

```
main()
{
    printf('hello ');
    printf('world');
    printf('\n');
}
```

C is a typed language — that is, each variable must be declared as a particular type. The following types are always available:

char — characters, occupying a single byte.
short — short integers.
int — normal integers.
long — long integers.
float — floating point (real) numbers.
double — double precision floating point numbers.

Variable declarations are made by giving the type name followed by a list of names of the variables that you want to be of that type. All variables must be declared, though as we'll see, the declarations can take place almost anywhere in the program. Additionally they may be any length, but usually only the first eight characters are significant.

ASSIGNMENT AND ARITHMETIC

As usual, spaces are not allowed in variable names but the underscore character can be used. Names must not begin with a digit, and should not begin with an underscore, but this is only because of a possible conflict with library functions. Case is normally significant — so that, for example, A is a different variable to a. The actual number of bytes for each numeric type is not specified, so that it may be different for each implementation. Typically, though, a short integer will be eight bits, int will be 16 bits and long will be 32 bits.

Assignment and arithmetic follow fairly normal conventions, with the equals sign (=) used as the assignment operator. The usual arithmetic operators +,–,* and / are used along with % which gives the modulus. Thus:

```
int x,y,z
. . .
z=x%y
```

Expressions and assignments can include any variety of numeric types, and char, which for these purposes is regarded as a one-byte integer containing the ASCII code for the character. C will automatically carry out any type conversions necessary by converting 'lower' to 'higher' types. It's worth noting here that all floating point operations are normally carried out in double precision, even when all the operands are merely float. It's also possible to force a change of type by means of a 'cast' whereby the variable name is preceded by the new type in brackets. For example:

```
int n;
float f;
. . . .
f=sqrt((double) n);
```

takes a double precision copy of the value in n since the square root function expects its argument to be of type double, but the actual value in n is unchanged.

Two very useful additional operators provided by C are the increment and decrement operators ++ and --. So, for example, ++a means increment and --a means decrement the value in a *before* the current statement is executed; a++ means increment and a-- means decrement the value in a *after* the current statement is executed. Thus:

```
int a,b
. . .
b = 1;
a = b++ ; /* leaves 1 in a and 2 in b */
```

Whereas:

```
b = 1
a = ++b :/* leaves 2 in both a and b*/
```

One further twist is that assignment, like all other operations in C, is treated like function and returns a value — namely, the value being assigned. This is a perfectly legitimate value to use in any other expression so, for example, we can write:

```
x=y=z;
```

which in effect assigns the value in z to both x and y.

We've already noted that input/output is handled by library functions and may vary between implementations. However, it's clearly difficult to write many programs without some form of I/O, so we'll round off this first instalment by briefly considering the two main functions for terminal I/O. We've already met one of these — printf, which will format and output values of various types onto the standard output device (normally the screen). Its full syntax is:

```
printf (control_string, arg1, arg2, . . . . .) ;
```

where the arguments may be strings or variables of any of the basic data types.

The control_string can contain ordinary sequences of characters, which are simply output as they are, but it also contains a format specifier for each of the arguments which, incidentally, also provides the number of arguments. A format specifier begins with a % sign and ends with one of the conversion characters, as shown in the Format Conversion table. Between them may appear –, to specify left justification in the output field, and a number to specify the field width in which the value is to be printed. This may be followed optionally by a point and another number specifying the number of characters to be actually printed (in the case of a string), or the number of places after the point (in the case of a float or double precision number).

Escape Sequences

\n	Newline
\t	Horizontal tab
\b	Backspace
\r	Carriage return
\f	Form feed
\'	Outputs the apostrophe character (otherwise used to surround a character)
\"	Outputs the quotation mark character (otherwise used to surround strings)
\d	Where d is an octal digit. Outputs the character with the corresponding ASCII code
\xh	Where h is a hex digit. Outputs the character with the corresponding ASCII code
\e	Escape
\v	Vertical tab

Format Conversion

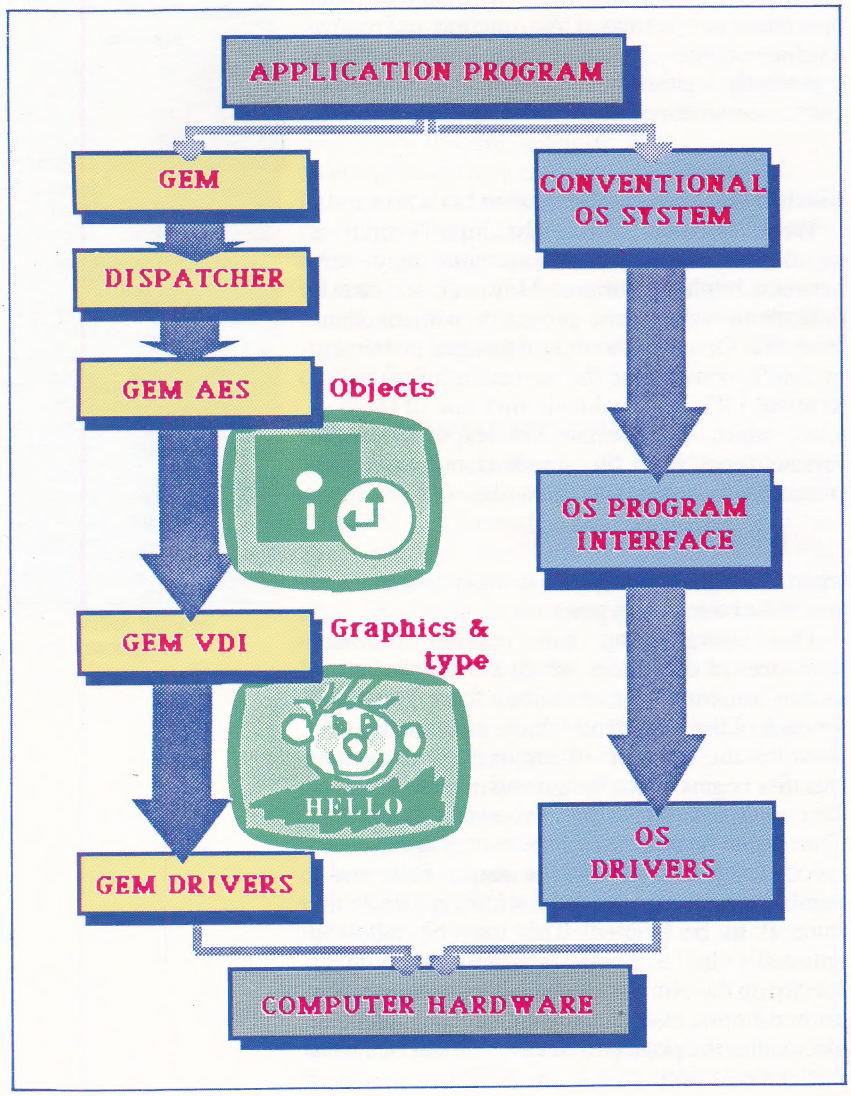
Character	Type Of Argument	Action
d	int	decimal format
o	int	octal format
x	int	hex format
u	int	unsigned decimal format
c	char/int	output as single character
s	*char	output as string
e	float/ double	mantissa-exponent format
f	float/ double	ordinary decimal format
g	float/ double	whichever of d, e or f is shortest
%		output the % character

WORKING WITH WINDOWS

We saw in the previous instalment how GEM achieves a high degree of portability by implementing as its core the international standard GKS 'shell'. We conclude our look at the GEM operating environment by considering a number of its operating facilities and program structures, as well as the possibilities for the serious user.

While Microsoft's Windows is likely to emerge as an important product for business computer systems, the home computer user is much more likely to find GEM in affordable form. Already GEM comes bundled with the new Motorola 68000-based Atari 520ST and is also appearing on machines such as the Apricot and RML Nimbus.

Different Routes
GEM's easy-to-use 'front-end' image belies its innate complexity. The VDI (virtual device interface) allows machine-independent code to be written, and is divided into two halves. One half, the GDOS, performs a near-traditional OS role processing user inputs, and the other half (the device drivers) is adapted to the particular hardware configuration being used. The AES (applications environment services) module contains various higher level subroutines for screen and peripheral handling



CAROLINE CLAYTON

Just as conventional applications programmers can write for one operating system, rather than a particular machine, GKS-based WIMPs provide a standardised way of addressing the underlying graphics devices. A program written for GEM, which is based on GKS, is notionally portable across any hardware that has the necessary software graphics kernel. Graphics kernel systems such as GEM, GSX and GKS act as a soft shell around both the existing system and the graphics hardware. Keyboard input and file management are controlled by the host operating system, as before, but any graphics handling is intercepted by the graphics 'operating system'.

Digital Research refers to its product as GEM DOS, but GEM itself does not actually perform any of the normal DOS functions. Commands from devices other than the keyboard (a mouse, for instance) are translated into equivalent calls to the existing system routines. The updating of the screen display or other graphics devices is handled by the underlying GSX system — the heart of GEM. This process is additional to the conventional system's command line interpreter.

Whenever a system operation requires display of data, the usual character by character display is replaced with a bit-mapped graphics data stream to the appropriate device driver. Each hardware device has its own driver with properties appropriate to that device's function (input, output, resolution, colour and so on). Device drivers can be installed as required. This soft plane of communication between the system and specific hardware devices is called the *virtual device interface* (VDI).

GEM PROGRAMMING

Programming for GEM itself is a complex process. In the absence of a powerful library of procedures to handle events, activate and de-activate windows (restoring any overlapped contents) and so on, the GEM programmer must build any required routine by calling the GEM/GSX VDI primitives. For example, the trivial task of creating a window and writing some text to it, as in BASIC's PRINT 'Hello!', entails a considerable amount of programming effort with GEM. First of all, the window position and size must be initialised to some default values. These must be changeable, should the window be 'dragged' or resized, and other attributes must be set such as text and background colours (intensities, if monochrome), character fount, size and so on.

The activation of another window is detected as an 'event', whereupon the current application is overlapped by this newly activated window. It may temporarily suspend processing until re-activated, but may also be required (with MS-Windows, Concurrent DOS and the new multitasking version of GEM) to carry on its processing in the background. When brought back to life, the process must display itself as the topmost window on the desk top and, significantly, it is responsible for re-drawing itself on top of other windows

displayed on screen.

Every program, no matter how trivial, must contain all the code for its own window-handling, including keeping track of the current contents of the window and its position relative to any undisplayed data. This in turn means an *overhead* of around 10 Kbytes of source code before a program starts to do anything.

There is no generalised GEM or GSX procedure library to call upon to handle these WIMP housekeeping chores. Software houses writing for GEM are programming in MODULA-2, PASCAL, C, BCPL and even assembler. But despite their expertise, it still poses problems, so home computer users should bear this in mind when thinking about writing simple GEM applications in a few hundred lines of BASIC. However, if you can program in any of the languages just mentioned, there are a number of products appearing that promise to provide a usable programming interface to the complexities of the underlying WIMP system.

WIMP TOOL KIT

The Bristol firm TDI was among the first to market a usable and affordable tool kit for WIMP programming. Called MODULA-2/ST, and selling for a modest £99, it offers Atari 520ST users a chance to do some serious programming for GEM in the language designed by Niklaus Wirth for Lilith (see page 1682).

As we've already seen, the software house Prospero supplies a library of routines for use with their high-quality PASCAL and FORTRAN compilers called Prospect. This provides a high-level interface to GSX — Digital Research's first subset implementation of GKS (see page 1704). The GSX system is incorporated into GEM, acting as the low-level virtual device interface — the heart of the system — and Prospect will also be available for GEM.

If you are interested in using GEM, however, several systems are possible for the serious user, including the following:

- An Atari 520ST (bundled with GEM), MODULA–2 and the TDI Toolkit, with an entry level price of just under £1,000.
- An Apricot (or, better still, RML Nimbus), GEM, a good quality MODULA–2, PASCAL, BCPL or C compiler costing over £1,000.
- A CP/M-86 or MS-DOS system with GSX, Pro-PASCAL and the Prospero Prospect graphics interface library. This is well tested, proven and available, but also expensive.
- An IBM PC, AT or 100 per cent compatible, an 8086 assembler or C, and the Digital Research GEM Programmer's Toolkit. This is useful for the very experienced with a thorough knowledge of systems programming and a lot of spare time and money.

For those who haven't the time to program, Digital Research has a suite of applications that run under the GEM environment. GEM Desktop is a desktop manager for running other applications (including multitasking on the new version of GEM), file handling, printing and plotting with desk accessories such as a clock and calculator. GEM Write is a WIMP-like word processor, and GEM Paint is a drawing package for artistic applications. These may be purchased individually for under £50, or as a three-pack called the GEM Collection. Unfortunately, the products available from DR will only run on 256 Kbyte IBM PCs (or absolute compatibles), and users of other machines will have to wait for their respective manufacturers to produce their own versions.

The Many Facets Of GEM

GEM contains two main functional parts: the 'applications environment services' (AES) and the 'virtual device interface' (VDI). The components of GEM AES include the subroutine libraries (which include the shells for windowing, mouse tracking, displaying messages and drawing objects) and a multitasking dispatcher — currently limited to three tasks, but with a capability of up to 12 in the new version of GEM. AES also incorporates the desk accessory buffer (for the clock, calculator and so on), as well as the menu/alert buffer (the event detector).

GEM VDI is essentially the 'graphics system extension' (GSX) core derived from GKS, but enhanced with RASTER-OPS and FACES. The former is the bitwise AND, OR and XOR operations on source and target blocks. The latter stores the character founts in dynamically loadable files.

These lower level facilities are separated into two parts. The first is GDOS (graphics device operating system), which is analogous to a normal OS interface. The second part, the installable device drivers, are dependent on each system's hardware configuration and peripheral devices.

All the basic host and device-independent graphics functions are contained in the GDOS, enabling it to be system-independent (device-specific details are all in the separate driver section). Just as a conventional operating system provides a consistent way for applications to access disk drives and so on, GDOS gives a standard and portable graphics system that remains constant, regardless of any specific workstation attributes.

The main entry point into the VDI is a single subroutine with five arguments:

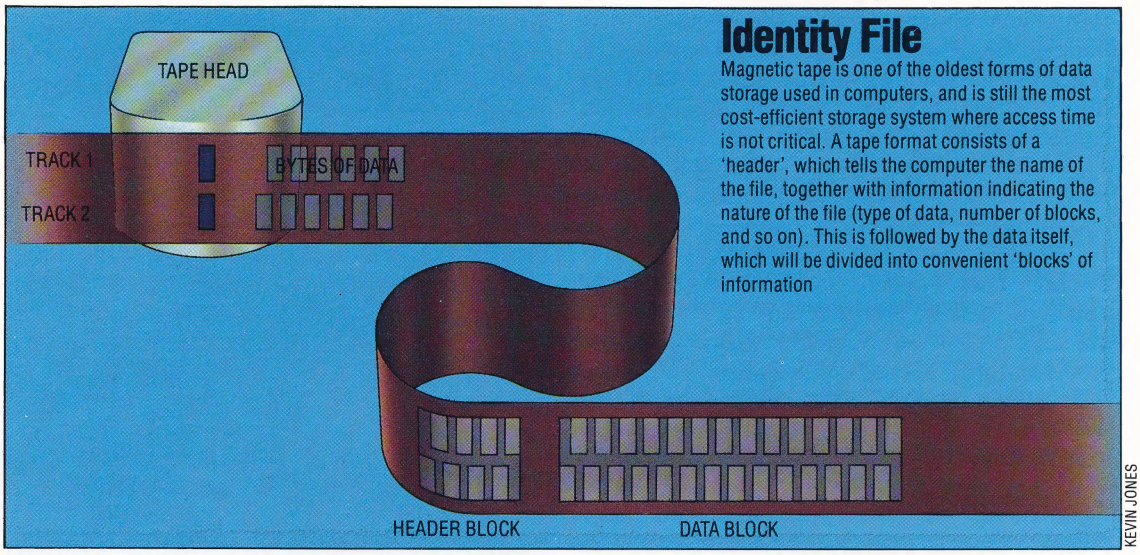
- Control array
- Array of input parameters
- Array of input point co-ordinates
- Array of output parameters
- Array of output point co-ordinates

The C language bindings have identifiers that act as mnemonic 'memos' for the complexity and sheer number of available calls. Thus, function calls, for example, beginning with v_ are all calls to VDI routines

TABLE

This is an array set aside within a program for the storage of data, and each element can be identified by a unique marker, known as a 'key'. A useful variation of the *table* is often used by machine code programmers for rapid access of data, and this is known as a 'look-up' table. Calculations which need to be done on data within a program can be time-consuming or difficult to perform in machine code. In this case, the computed values are held in a table that can be accessed using the value on which the calculation is to be made.

TAPE FORMAT


TAPE HEAD
TRACK 1
TRACK 2
BYTES OF DATA
HEADER BLOCK
DATA BLOCK
Identity File
Magnetic tape is one of the oldest forms of data storage used in computers, and is still the most cost-efficient storage system where access time is not critical. A tape format consists of a 'header', which tells the computer the name of the file, together with information indicating the nature of the file (type of data, number of blocks, and so on). This is followed by the data itself, which will be divided into convenient 'blocks' of information
KEVIN JONES

The *tape format* is still the most commonly used method of storing and accessing information on home computers. Most machines have their own unique method of storing programs and other data on tape, although the basic principles remain the same.

Typically, a collection of data will begin with a tape header, which is information telling the computer something about the data. This will include such details as the name of the file, the length of the program or data, whether it is a sequential or program file and so on. This will then be followed by the data itself. Usually, although by no means always, the data will be divided into blocks, each of which will be delimited by tape marks that tell the computer a block of data is about to start or has finished.

Tape format can also refer to the way in which the electromagnetic fields from the tape head are encoded onto the tape itself.

TASK

A *task* is performed by a computer, consisting of the data and files required to complete the task and the program which acts upon the data. Sometimes a number of separate programs need to be run consecutively, often calling each other. This collection of programs can also be considered together as one task.

One of the features that is becoming popular on many of the more expensive business machines is 'multitasking'.This means that the computer can hold a number of tasks in its memory and devote time to each in turn. For example, if a computer is being used for word processing but is inactive for a moment, it may switch to data processing 'in the background'. This is not to be confused with parallel processing, in which a number of tasks may be performed simultaneously.

TELECONFERENCING

Allowing people in different cities to hold a 'meeting' as though they were all in the same room, a *teleconferencing* system requires that each person has access to a computer terminal that can transmit and receive messages. When information is typed in at one terminal, the 'manager' — a system that controls the proceedings — transmits the data to each terminal VDU involved in the conference.

Recently, the system has been updated to include the transmission of pictures via a video link and microphones to transmit speech along the standard telephone network. Furthermore, the manager can keep a log of the transmissions and provide those attending the 'conference' with a full transcript.

Although basically a good idea, since it means that businessmen can hold conferences without having to travel, teleconferencing has been slow to take off.

While teleconferencing over large distances results in a second's delay or so, the situation becomes much more difficult when transmitting pictures, which require large quantities of data to maintain an effective image. But for some applications, such as research teams working on the same project in different parts of the world, teleconferencing is a useful and time-saving means of communication.

The concept of teleconferencing, however, remains unproved. Although further reductions in the operating costs of using such systems would doubtless lead to an increase in its use, many businessmen have found that the teleconference lacks the immediacy of a face to face meeting.

CARDS UP!

We began our Pontoon project by looking at card display and shuffling routines for the Commodore 64 (see page 1712). We now turn out attention to versions of the same routines for the Amstrad, Spectrum and BBC micros. The principal differences are the need to define characters for the card suits and borders, and a problem with the TAB function for these micros. Tabbing to a certain position on a line erases all data to the left of that TAB position. We therefore need to adapt our display routine slightly.

Card Display And Shuffling

Amstrad CPC Range

Main Program:

```
10 REM **** Amstrad Pontoon ****
20 GOSUB 500:REM init array et
50 REM **** Game Loop ****
55 GOSUB 600:REM init game
```

Array Initialisation:

```
500 REM **** init arrays etc ****
505 INK 0,16:pink=0:INK 1,3:red=1:INK 2,
0:black=2:INK 3,26:white=3
512 bk$=STRING$(7,207)
513 br$=CHR$(149):li$=STRING$(7,154)
515 DIM x(2),y(2):REM next card position
s
520 su$=CHR$(228)+CHR$(227)+CHR$(226)+CH
R$(229):REM suit symbols
530 DIM cn$(13):FOR i=1 TO 13:READ cn$(i
):NEXT i:REM read number data
540 DIM cd$(13):FOR i=1 TO 13:READ cd$(i
):NEXT i:REM read pattern data
560 DIM dk(52,2):REM card deck
570 GOSUB 3000:REM shuffle pack
580 BORDER 9:PAPER pink:REM screen colou
rs
590 RETURN
600 REM **** init game ****
605 CLS
610 hp(1)=1:hp(2)=1
620 x(1)=0:y(1)=0:x(2)=20:y(2)=0
625 bt=0:sb=0:GOSUB 4200:REM print bet b
ox
630 RETURN
```

Card Display Routine:

```
900 REM **** print at ****
910 LOCATE tx+1,ty+1:RETURN
1000 REM **** display card ****
1010 GOSUB 1050:GOSUB 1100:RETURN
1050 REM **** card blank ****
1055 tx=x(pl):ty=y(pl):GOSUB 900:REM pos
ition
1060 PEN white:PRINT CHR$(150);li$;CHR$(
156)
1070 FOR i=1 TO 9:tx=x(pl):ty=ty+1:GOSUB
 900:PRINT br$;SPACE$(7);br$:NEXT i
1080 tx=x(pl):ty=ty+1:GOSUB 900:PRINT CH
R$(147);li$;CHR$(153)
1090 RETURN
1100 REM **** card detail ****
1120 tx=x(pl)+2:ty=y(pl)+1:GOSUB 900:REM
 position
1125 ct$=MID$(su$,su,1):REM select suit
type
1127 co=red:IF su>2 THEN co=black
1128 PEN co
1130 FOR i=1 TO 19 STEP 3
1140 cc$=MID$(cd$(cn),i,3):cl$=""
1142 FOR j=1 TO 3:c$=SPACE$(2)
1144 IF MID$(cc$,j,1)="1" THEN c$=ct$+SP
ACE$(1)
1146 cl$=cl$+c$:NEXT j
1150 tx=x(pl)+2:ty=ty+1:GOSUB 900:PRINT
cl$:NEXT i
1160 REM **** add corner labels ****
1170 tx=x(pl)+1:ty=y(pl)+1:GOSUB 900:PRI
NT cn$(cn):REM number
1180 ty=ty+1:GOSUB 900:PRINT ct$:REM sui
t
1190 tx=x(pl)+7:ty=y(pl)+9:GOSUB 900:PRI
NT cn$(cn):REM bottom number
1192 x(pl)=x(pl)+2:y(pl)=y(pl)+1
1195 RETURN
1200 REM **** display card back ****
1210 GOSUB 1050:GOSUB 1250:RETURN
1250 REM **** card back ****
1255 tx=x(pl):ty=y(pl):GOSUB 900:REM pos
ition
1260 FOR i=1 TO 9:tx=x(pl):ty=ty+1:GOSUB
 900:PRINT br$;:PEN red:PRINT bk$;:PEN w
hite:PRINT br$:NEXT i
1270 x(pl)=x(pl)+2:y(pl)=y(pl)+1
1280 RETURN
1300 REM **** deal a card ****
1310 cn=dk(dp,1):su=dk(dp,2)
1320 dp=dp+1:IF dp>52 THEN dp=1:REM bump
 deck
1330 IF fl=1 THEN GOSUB 1200:RETURN:REM
display back of card
1335 GOSUB 1000:RETURN:REM display card
2000 REM **** card number data ****
2010 DATA A,2,3,4,5,6,7,8,9,T,J,Q,K
2100 REM **** card display data ****
2110 DATA "00000000010000000000"
2120 DATA "00001000000000010000"
2130 DATA "00001000001000010000"
2140 DATA "00010100000000101000"
2150 DATA "00010100001000101000"
2160 DATA "00010100010100101000"
2170 DATA "00010101010100101000"
2180 DATA "00010101010101101000"
2190 DATA "10100010101010000101"
2200 DATA "10101010100010101010"
2210 DATA "00000000001000000000"
2220 DATA "00000000010000000000"
2230 DATA "00000000010000000000"
```

Shuffling The Deck:

```
3000 REM **** shuffle the deck ****
3005 RANDOMIZE TIME:dp=1
3007 FOR i=1 TO 52:dk(i,1)=0:NEXT i
3010 FOR i=1 TO 4:FOR j=1 TO 13
3020 ep=INT(RND(1)*52)+1:REM select entr
y point
3030 IF dk(ep,1)=0 THEN 3050
3040 ep=ep+1:IF ep>52 THEN ep=1
3045 GOTO 3030
3050 dk(ep,1)=j:dk(ep,2)=i:NEXT j,i
3060 RETURN
```

BBC Micro

Note: Type PAGE=&0E00 before typing in the program or loading it from disk or tape

Main Program:

```
10 REM BBC PONTOON
15 MODE 1
20 GOSUB 500
50 REM
55 GOSUB 600
```

Array Initialisation:

```
500 REM
510 SP$="":FOR I=1 TO 39:SP$=SP$+" ":N
EXT I
512 BK$="":LI$="":FOR I=1 TO 7:LI$=LI$
```

```
+CHR$(195):BK$=BK$+CHR$(166):NEXT I
  513 BR$=CHR$(194)
  515 DIM X(2),Y(2)
  520 SU$=CHR$(211)+CHR$(218)+CHR$(216)+
CHR$(193)
  530 DIM CN$(13):FOR I=1 TO 13:READ CN$
(I):NEXT
  535 CN$(10)=CHR$(128)
  540 DIM CD$(13):FOR I=1 TO 13:READ CD$
(I):NEXT
  560 DIM DK(52,2)
  570 GOSUB 3000
  571 COLOUR 131:COLOUR 0
  572 VDU 23,166,165,66,90,36,90,165,90,
66
  573 VDU 23,195,0,0,0,255,255,0,0,0
  574 VDU 23,194,24,24,24,24,24,24,24,24
  575 VDU 23,193,16,56,124,254,254,238,8
4,56
  576 VDU 23,216,24,60,90,255,255,90,24,
60
  577 VDU 23,218,16,56,124,254,254,124,5
6,16
  578 VDU 23,211,36,126,255,255,255,126,
60,24
  579 VDU 23,213,0,0,0,15,31,28,24,24
  580 VDU 23,201,0,0,0,240,248,56,24,24
  581 VDU 23,202,24,24,28,31,15,0,0,0
  582 VDU 23,203,24,24,56,248,240,0,0,0
  583 VDU 23,128,206,219,219,219,219,219
,206,0
  590 RETURN
  600 REM
  601 DB=0:CS=0
  605 CLS
  620 X(1)=0:Y(1)=0:X(2)=20:Y(2)=0
  630 RETURN
```

Card Display Routine:

```
  900 REM **** PRINT AT ****
  910 PRINT TAB(TX,TY);:RETURN
 1000 REM
 1010 GOSUB 1050:GOSUB 1100:RETURN
 1050 REM
 1055 TX=X(PL):TY=Y(PL):GOSUB 900:REM PO
SITION
 1060 COLOUR 0:PRINT TAB(TX,TY);CHR$(213
);LI$;CHR$(201)
 1070 FOR I=1 TO 9:PRINT TAB(TX,TY+I);BR
$;"        ";BR$:NEXT I
 1080 PRINT TAB(TX,TY+10);CHR$(202);LI$;
CHR$(203)
 1090 RETURN
 1100 REM
 1120 TX=X(PL)+2:TY=Y(PL)+1:GOSUB 900
 1125 CT$=MID$(SU$,SU,1)
 1127 COLOUR 1:IF SU>2 THEN COLOUR 0
 1130 FOR I=1 TO 19 STEP 3
 1140 CC$=MID$(CD$(CN),I,3):CL$=""
 1142 FOR J=1 TO 3:C$=CHR$(9)+CHR$(9)
 1144 IF MID$(CC$,J,1)="1" THEN C$=CT$+C
HR$(9)
 1146 CL$=CL$+C$:NEXT J
 1150 PRINT TAB(X(PL)+2,TY+((I+3)/3));CL
$:NEXT I
 1170 TX=X(PL)+1:TY=Y(PL)+1:GOSUB 900:PR
INT CN$(CN)
 1180 TY=TY+1:GOSUB 900:PRINT CT$
 1190 TX=X(PL)+7:TY=Y(PL)+9:GOSUB 900:PR
INT CN$(CN)
 1192 X(PL)=X(PL)+2:Y(PL)=Y(PL)+1
 1195 RETURN
 1200 REM
 1210 GOSUB 1050:GOSUB 1250:RETURN
 1250 REM
 1255 TX=X(PL):TY=Y(PL)+1:GOSUB 900
 1260 FOR I=1TO9:PRINT TAB(TX,I);BR$;BK$
;BR$:NEXT I
 1270 X(PL)=X(PL)+2:Y(PL)=Y(PL)+1
 1280 RETURN
 1300 REM
 1310 CN=DK(DP,1):SU=DK(DP,2)
 1320 DP=DP+1:IF DP>52 THEN DP=1
 1330 IF FL=1 THEN GOSUB 1200:RETURN
 1335 GOSUB 1000:RETURN
 2010 DATA A,2,3,4,5,6,7,8,9,T,J,Q,K
 2110 DATA"00000000010000000000"
 2120 DATA"00001000000000010000"
 2130 DATA"00001000010000010000"
 2140 DATA"00010100000000101000"
 2150 DATA"00010100010000101000"
 2160 DATA"00010100010100101000"
 2170 DATA"00010101010100101000"
 2180 DATA"00010101010101101000"
 2190 DATA"10100010101010100101"
 2200 DATA"10101010100010101101"
 2210 DATA"00000000010000000000":REM J
 2220 DATA"00000000010000000000":REM Q
 2230 DATA"00000000010000000000":REM K
```

Shuffling The Deck:

```
3000 REM
3005 R=RND(-TIME):DP=1
3007 FOR I=1 TO 52:DK(I,1)=0:NEXT I
3010 FOR I=1 TO 4:FOR J=1 TO 13
3020 EP=INT(RND(1)*52)+1
3030 IF DK(EP,1)=0 THEN 3050
3040 EP=EP+1:IF EP>52 THEN EP=1
3045 GOTO 3030
3050 DK(EP,1)=J:DK(EP,2)=I:NEXT J,I
3060 RETURN
```

Sinclair Spectrum

Main Program:

```
 10>REM **** SPECTRUM PONTOON ****
 15 PRINT "PLEASE WAIT..."
 16 POKE 23658,8
 20 GO SUB 500: REM INIT ARRAYS ETC
 50 REM **** GAME LOOP ****
 55 GO SUB 600: REM INIT GAME
```

Array Initialisation:

```
 500>REM **** INIT ARRAYS ETC ****
 510 LET S$="": FOR I=1 TO 25: LET S$=S$
+" ": NEXT I
 511 LET S$=S$+S$( TO 14)
 512 LET B$="": LET L$="": FOR I=1 TO 7:
 LET L$=L$+CHR$ 145: LET B$=B$+CHR$ 144:
 NEXT I
 513 LET D$=CHR$ 146
 515 DIM X(2): DIM Y(2): REM NEXT CARD P
OSITIONS
 520 LET U$=CHR$ 147+CHR$ 148+CHR$ 149+C
HR$ 150: REM SUIT TYPES
 530 DIM N$(13): FOR I=1 TO 13: READ N$(
I): NEXT I: REM READ NUMBER DATA
 540 DIM C$(13,21): FOR I=1 TO 13: READ
C$(I): NEXT I: REM READ PATTERN DATA
 560 DIM K(52,2): REM SET UP CARD DECK A
RRAY
 570 GO SUB 3000: REM SHUFFLE PACK
 580 BORDER 4: PAPER 7: INK 9: CLS
 581 REM **** SET UP UDG'S ****
 582 FOR L=USR "A" TO USR "L"+7
 583 READ US: POKE L,US: NEXT L
 590 RETURN
 600 REM **** INIT GAME ****
 602 LET DB=0: LET CS=0
 605 CLS
 620 LET X(1)=0: LET Y(1)=0: LET X(2)=14
: LET Y(2)=0
 630 RETURN
```

Card Display Routine:

```
  900>REM **** PRINT AT ****
  910 PRINT AT TY,TX;: RETURN
 1000 REM **** DISPLAY CARD ****
 1010 GO SUB 1050: GO SUB 1100: RETURN
 1050 REM **** CARD BLANK ****
 1055 LET TX=X(PL)+1: LET TY=Y(PL): GO SU
B 900: REM POSITION
 1060 PRINT AT Y(PL),TX;CHR$ 151;L$;CHR$
 152
 1070 FOR I=1 TO 9: PRINT AT Y(PL)+I,TX;C
HR$ 146;"        ";CHR$ 146: NEXT I
 1080 PRINT AT 10+Y(PL),TX;CHR$ 153;L$;CH
R$ 154
 1090 RETURN
 1100 REM **** CARD DETAIL ****
 1120 LET TX=X(PL)+2: LET TY=Y(PL)+2: GO
 SUB 900: REM POSITION
 1125 LET T$=U$(SU): REM SELECT SUIT TYPE
```

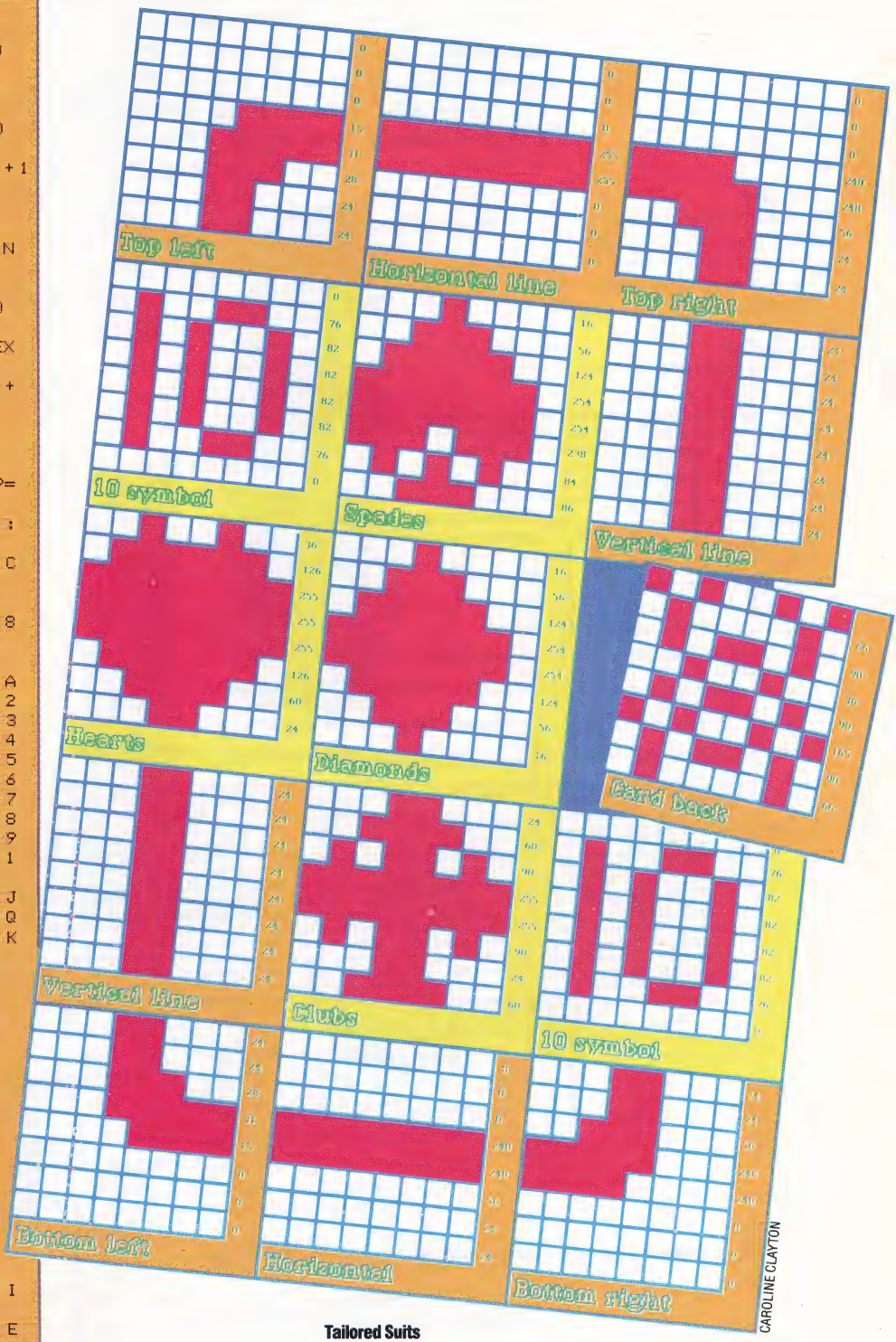
```
1127 INK 0: IF SU>2 THEN  INK 2: REM SEL
ECT COLOUR
1130 FOR I=1 TO 19 STEP 3
1140 LET Z$=C$(CN)(I TO I+2)
1142 FOR J=1 TO 3: LET W$="  "
1143 PRINT " ";
1144 IF Z$(J)="1" THEN  PRINT T$;
1145 IF Z$(J)="0" THEN  PRINT " ";
1146 NEXT J
1150 PRINT AT ((I+3)/3)+TY,X(PL)+2;: NEX
T I
1160 REM **** ADD CORNER LABLES ****
1170 LET TX=X(PL)+2: LET TY=Y(PL)+1: GO
SUB 900: PRINT N$(CN): REM POSITION
1180 LET TY=TY+1: GO SUB 900: PRINT T$:
REM SUIT
1190 LET TX=X(PL)+8: LET TY=Y(PL)+9: GO
SUB 900: PRINT N$(CN): REM POSITION
1192 LET X(PL)=X(PL)+2: LET Y(PL)=Y(PL)+1
1193 INK 0
1195 RETURN
1200 REM **** DISPLAY CARD BACK ****
1210 GO SUB 1050: INK 1: GO SUB 1250: IN
K 0: RETURN
1250 REM **** CARD BACK ****
1255 LET TX=X(PL)+2: LET TY=Y(PL)+1: GO
SUB 900: REM POSITION
1260 FOR I=1 TO 9: PRINT AT I,TX;B$: NEX
T I
1270 LET X(PL)=X(PL)+2: LET Y(PL)=Y(PL)+
1
1280 RETURN
1300 REM **** DEAL A CARD ****
1310 LET CN=K(DP,1): LET SU=K(DP,2)
1320 LET DP=DP+1: IF DP>52 THEN  LET DP=
1: REM BUMP DECK
1330 IF FL=1 THEN  GO SUB 1200: RETURN :
 REM DISPLAY BACK OF CARD
1335 GO SUB 1000: RETURN : REM DISPLAY C
ARD
2000 REM **** CARD NUMBER DATA ****
2010 DATA "A","2","3","4","5","6","7","8
","9",CHR$ 155,"J","Q","K"
2100 REM **** CARD DISPLAY DATA ****
2110 DATA "00000000001000000000": REM A
2120 DATA "00001000000000000000": REM 2
2130 DATA "00001000001000001000": REM 3
2140 DATA "00010100000000101000": REM 4
2150 DATA "00010100001000101000": REM 5
2160 DATA "00010100010100101000": REM 6
2170 DATA "00010101010100101000": REM 7
2180 DATA "00010101010101101000": REM 8
2190 DATA "10100010101010100101": REM 9
2200 DATA "10101010100010101010": REM 1
0
2210 DATA "00000000001000000000": REM J
2220 DATA "00000000001000000000": REM Q
2230 DATA "00000000001000000000": REM K
2240 REM **** UDG DATA ****
2250 DATA 165,66,90,36,90,165,90,66
2251 DATA 0,0,0,255,255,0,0,0
2252 DATA 24,24,24,24,24,24,24,24
2253 DATA 16,56,124,254,254,238,84,56
2254 DATA 24,60,90,255,255,90,24,60
2255 DATA 16,56,124,254,254,124,56,16
2256 DATA 36,126,255,255,255,126,60,24
2257 DATA 0,0,0,15,31,28,24,24
2258 DATA 0,0,0,240,248,56,24,24
2259 DATA 24,24,28,31,15,0,0,0
2260 DATA 24,24,56,248,240,0,0,0
2261 DATA 0,76,82,82,82,82,76,0
```

Shuffling The Deck:

```
3000>REM **** SHUFFLE THE DECK ****
3005 RANDOMIZE : LET DP=1
3007 FOR I=1 TO 52: LET K(I,1)=0: NEXT I
3010 FOR I=1 TO 4: FOR J=1 TO 13
3020 LET EP=INT (RND*52)+1: REM SELECT E
NTRY POINT
3030 IF K(EP,1)=0 THEN  GO TO 3050
3040 LET EP=EP+1: IF EP>52 THEN  LET EP=
1
3045 GO TO 3030
3050 LET K(EP,1)=J: LET K(EP,2)=I: NEXT
J: NEXT I
3060 RETURN
```

On The Border

All these versions of pontoon use character graphics to draw the card border. In the BBC Micro and Spectrum versions we have to define border characters for ourselves. In the BBC Micro version, all the suit and border definitions are done using the VDU 23 command between lines 572 and 583. In the Spectrum version, the character definitions are held as DATA between lines 2240 and 2261 and READ as part of the initialisation subroutine at lines 581 to 583



Tailored Suits

While the Commodore 64 and Amstrad CPC range of micros have card suit characters as part of their standard character sets, we need to define our own suit characters for the BBC Micro and Spectrum versions

TO AND FRO

While the RAM, ROM and central processing unit constitute the heart of a computer system, communication with the real world can't be achieved without the implementation of input/output facilities. We concentrate here on serial and parallel I/O and take a look at the PIO chip.

So far in this series, we've discussed the processor and how it communicates with ROM and RAM. The system we've described is self-contained, which is, of course, only of limited use. The basic memory/processor communicates with other devices in many ways. Input can be achieved using a keyboard, joystick or through some external device attached to the system via a port, and output is normally made to a VDU or printer.

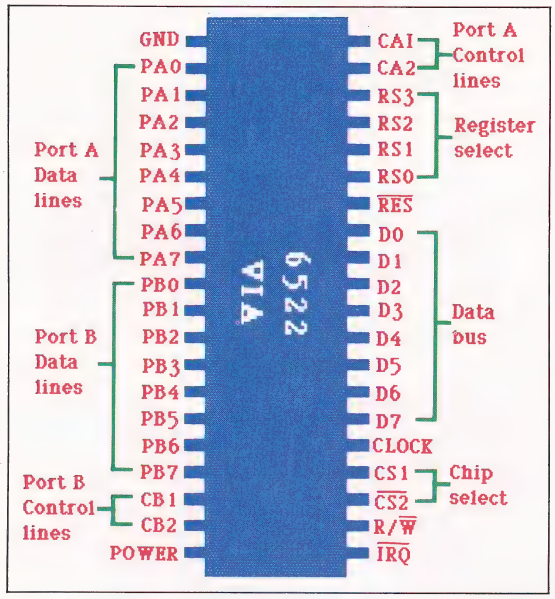
We can't connect I/O devices directly to the main data bus because output data is not normally present on it long enough for an external device to use it. Conversely, incoming data must be held valid on the data bus long enough for the processor to deal with it. At the barest minimum, therefore, the task of an I/O interface is to freeze data long enough for the input or output to be made and to synchronise I/O operations, normally using a handshaking procedure.

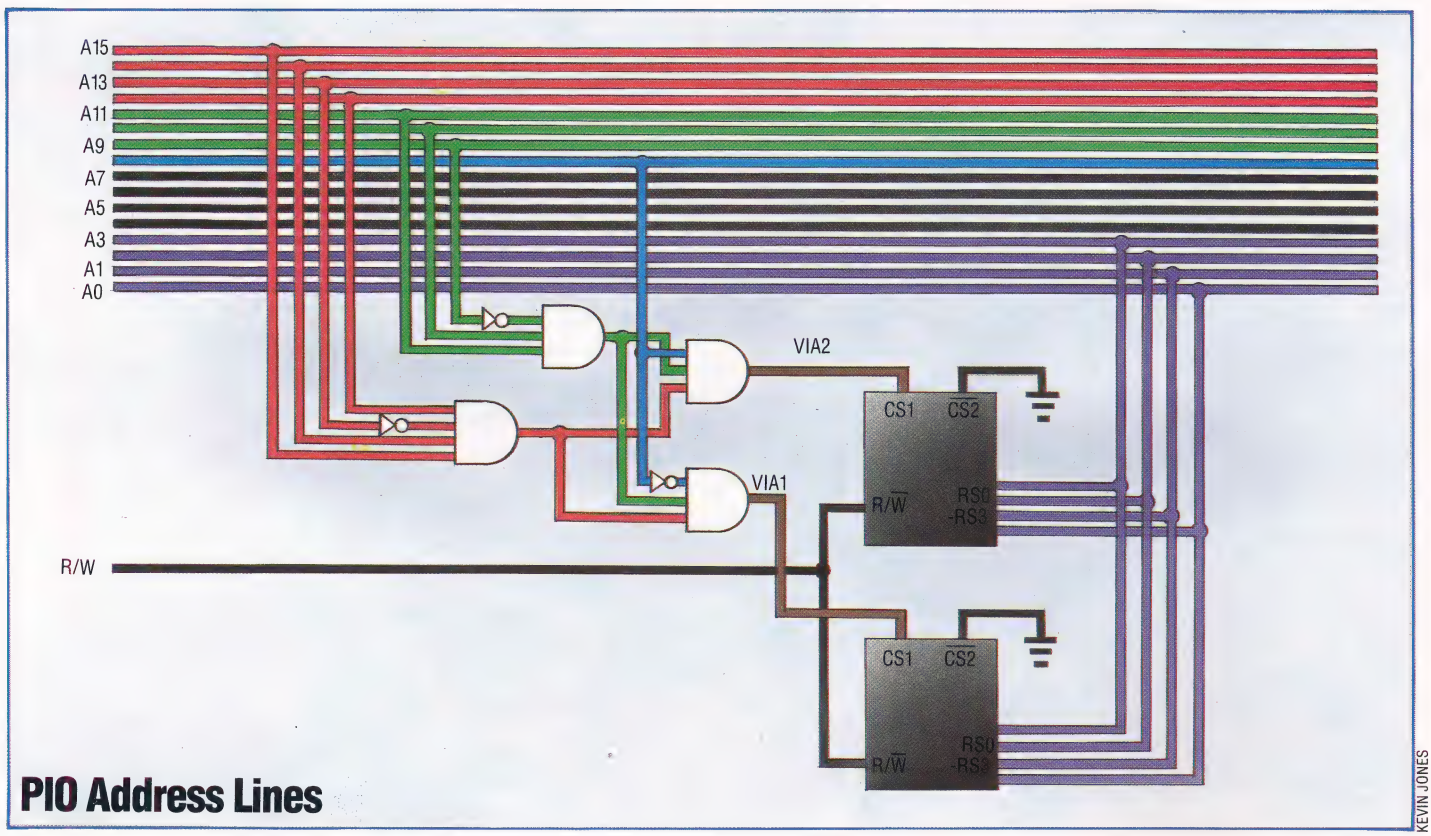
Most I/O devices, of course, do much more than provide a couple of latches and control lines. But before proceeding, it's worth making the distinction between the two main types of I/O device found in most home micros. If we ignore the problem of VDU display output, which is usually handled through specialised circuitry (because of the speed of data transfer required to update a screen, among other things) the type of I/O interface used will normally be dependent on the method by which the devices are attached — which will be either a serial or parallel link. I/O devices, therefore, fall into two categories — PIO and SIO devices.

We've already looked at serial methods of communications in detail. In particular, our MIDI

Pin In/Outs
The 6522 VIA chip used in the BBC Micro is a typical example of a parallel I/O chip — it has two eight-bit ports, and lines to the data bus

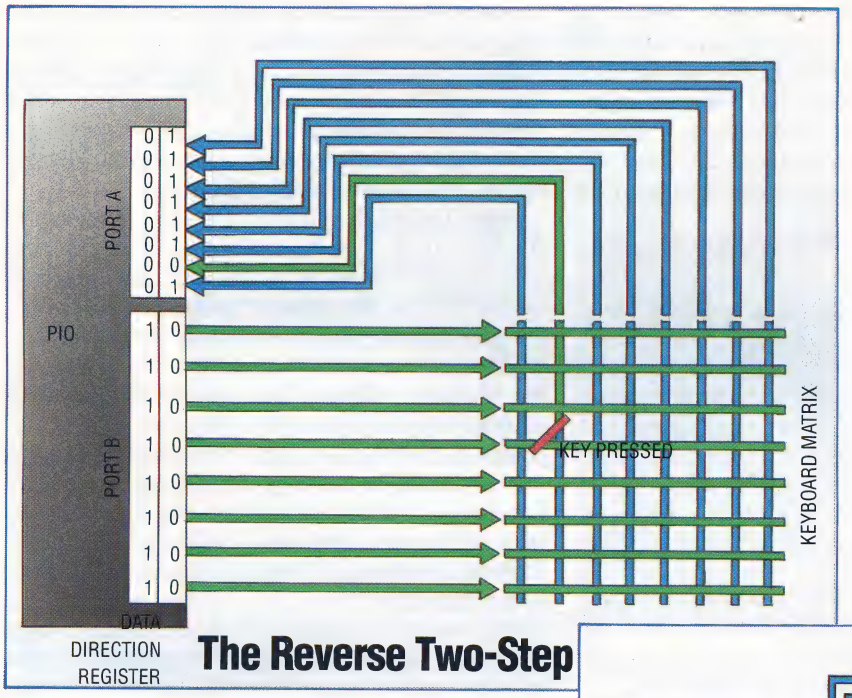
The PIO Code
PIO chips can be programmed by placing values into their internal registers. To allow access to these registers, the processor must be able to 'see' them in its own address space. The example here shows two PIO chips memory-mapped to appear at $DC00 and $DD00. The upper seven bits of the address bus are decoded and A8 selects $DC00 or $DD00. As each PIO has 16 internal registers, the lowest four bits from the address bus are used to address an individual register





PIO Address Lines

KEVIN JONES



Key To The Problem
The line reversal technique is one of several methods used to identify keypresses on the keyboard, and uses the directional programmability of the two eight-bit PIO Ports to output zeros along the keyboard matrix rows and then output the result received in the other port back along the columns. The resulting 16-bit code held in the two port data registers uniquely identifies the key pressed

The Reverse Two-Step

interface series in Workshop (see page 1335) provides a classic example of how to implement an asynchronous communications device. The essential components of such an interface are shift registers, which enable the parallel data bits from the data bus to be converted into a serial stream and an input stream to be converted to parallel format. In addition, automatic parity checking and generation (see page 186) is normally handled by this device.

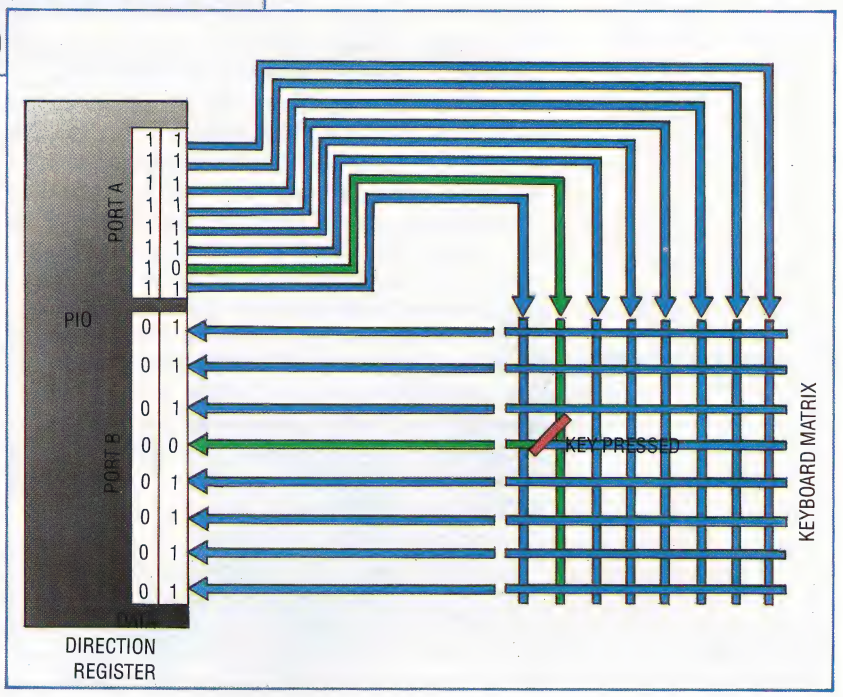
More immediately important to any home computer system is the PIO chip (see page 1228). The term PIO, however, is not universally used and terms such as VIA (versatile interface adaptor), PIA (peripheral interface adaptor) and CIA (complex interface adaptor) all refer essentially to the same device.

Home micros with full interfacing facilities, such as the BBC Micro and Commodore 64, are equipped with two PIO chips, one of which is normally dedicated to the on-board I/O devices like the keyboard and joystick ports. The second PIO can be used in conjunction with a parallel printer and user port, giving the user or third party hardware manufacturers access to the I/O system.

The first diagram shows the 6522 VIA chip used in the BBC Micro. Having 40 pins, this package provides eight data bus links and two eight-bit ports that can be used for input or output. Other lines include interrupt and read/write controls, register select bits and handshaking lines for the two I/O ports.

THE PIO CHIP'S STRUCTURE

Although a PIO chip is an extremely complex device electronically, it's reasonably easy to understand conceptually. Typically, a PIO chip found within an eight-bit micro will have two I/O ports — A and B — with individual data direction control on each port bit provided by data direction registers and separate handshaking control lines. Many PIOs also include a shift register that can be used to transmit or receive data along one of the port handshaking lines. These will usually be under internal timing control from one of the two timers, or under an external timing source input via another handshaking line.



An eight-bit interrupt status register uses each bit to indicate an interrupt request from a number of sources — the handshaking control lines, timers or shift registers. A programmable interrupt enable register allows the programmer to select which (if any) of the PIO interrupt sources actually need to interrupt the processor.

Having outlined the basic functions of a PIO, let's turn our attention to how it can be linked into the processor/memory system. Most systems with full I/O facilities employ two PIO chips. Let's suppose we wanted the internal registers of these

two devices to appear at $DC00 and $DD00. In order to pass data to and receive data from the PIOs, the processor must be able to treat the PIO internal registers as though they were memory locations and address them accordingly. To make the PIO registers appear in the address space, we must therefore tap off and gate some lines from the address bus.

The diagram shows how the highest eight bits from the address bus are gated to select either PIO. The upper four bits are gated to produce a one when 1101 ($D) is present. The next three bits are gated to produce a one when 110 is present. Selection between $DC00 and $DD00 is actually made by address bit A8. The lowest four bits connect directly to the four register select pins on the PIO. Thus the internal registers of PIO 1 can be addressed as locations $DC00 to $DC0F, and those of PIO 2 addressed as $DD00 to $DD0F.

Twin Sentinels
The diagram here shows how two PIO chips can be used to interface a keyboard, parallel printer, joysticks and user port to the data bus. One of the main tasks of the firmware or kernel routines that are provided with the system is to program the PIOs to handle incoming and outgoing data. For the most applications, therefore, the machine code programmer can call the kernel routines rather than design his own. Sometimes, however, it is useful to program the PIO directly, such as to use the PIO timers or to utilise the user port

CONNECTING A KEYBOARD

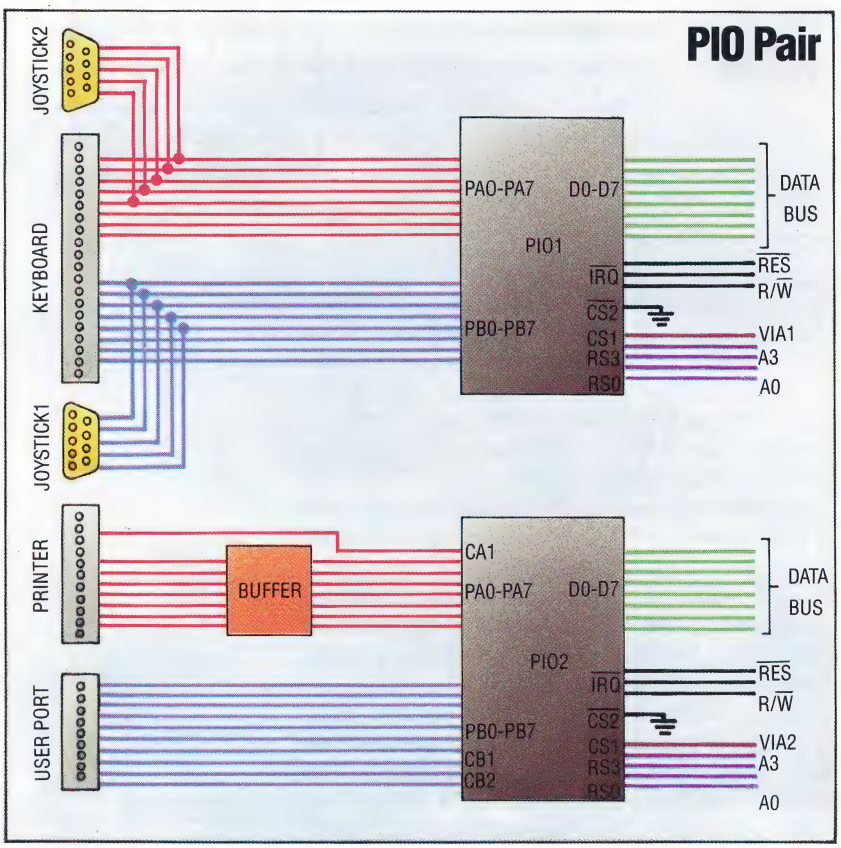
One PIO is normally dedicated to the system, for connecting joysticks and the main computer keyboard. Several methods of connecting a complex 64-key keyboard exist, the simplest of which is to construct the keyboard so that there is an underlying matrix of wires, arranged in eight rows and eight columns. The row lines are then connected to one PIO port and the column lines are connected to another, as shown in the diagram. When a key is pressed, the corresponding column and row wires at the intersection are connected.

The method of line reversal used to detect which key has been pressed is mainly software-based, relying upon the PIO port's ability to be programmed for input or output. Step 1 involves setting up port B to output and placing zero in its data register. Port A is set up for input and, since the column lines are all held high, the bit in port A's data register corresponding to the pressed key will be low, while all the others will be high.

Step 2 involves reversing the data direction of the two ports. Now port B acts as the input port and each bit will be set to one, with the exception of the bit that corresponds to the key row being pressed. The 16-bit code now held in the two port registers can be used to identify the location within the character ROM that contains the eight-bit code of the character whose key was pressed.

Other methods of key identification include row-scanning. In this case each column is set high, looking in turn for a corresponding high level on one of the row lines and dedicated chips that decode the keyboard matrix to generate character codes directly.

We've already looked at PIO address decoding. The final diagram shows how a pair of PIO chips may be used to access the keyboard and provide parallel printer and user ports. While the first may be taken up completely by connection to the keyboard, the second PIO has two eight-bit ports available. The first of these can be used as an output port for a printer, via a suitable buffer to protect the PIO against incorrect connections at the printer port. The second is then available as a user port. We've seen in some of our Workshop projects how this port can be programmed to control floor robots and jointed arms.



Programming The PIO

It's important to bear in mind that PIO chips are programmable — that is, the chip has several internal registers that can be accessed by the main processor via its normal addressing system. By placing certain values within these registers, the PIO can be made to behave in certain ways. For example, each of the eight data lines that make up each I/O port can be programmed independently to act as input or output lines by means of a data direction register. Thus a single port can be made to input and output simultaneously using different bits. A PIO can therefore be applied to most interfacing problems by designing suitable driving software.

In addition, since I/O operations have to be made at closely defined intervals, PIOs feature 16-bit countdown timers that can be loaded with a value and decremented with each pulse of the system clock, generating an interrupt signal when 'underflow' (the counter tries to decrement from zero) of the timer occurs.

Synchronous data communications, by which data is transmitted or received at regular intervals, can therefore be controlled through the PIO timers. Machine code games programmers often program the PIO timers to provide a regular source of interrupts for their own purposes

SQUEEZED IN

Word processing packages for home micros are invariably shaped by the potential of each individual machine. In particular, the range of features provided in any given program will depend on the amount of memory available. We look at three of the more successful packages for the most popular home computers.

Word processing is undoubtedly one of the most popular applications for home microcomputers. Programs range from simple text editors priced at a few pounds to sophisticated and expensive disk or ROM-based word processing packages. However, any program is only as good as the hardware it's designed for, and the packages written for most home micros don't have many of the more complex features provided on programs for business computers.

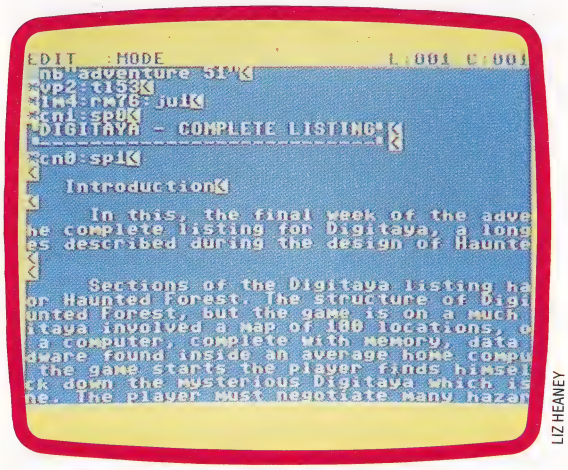
The primary restriction imposed on a word processor for an eight-bit computer is the limitations of memory. Thus every additional feature included in the software will take up space that would otherwise be available for storing text. This has proved to be a major stumbling block in the implementation of WordStar on the Amstrad CPC 464 and 664. Although these machines can use the CP/M operating system, there is insufficient memory available to adequately implement the full WordStar program. However, a scaled down version of the program — called Pocket WordStar — has been developed.

Another problem confronting word processor programmers of home micros is the method of back-up storage. Most home micros still rely on cassettes for back-up storage; thus, all of the program and text has to be held within the computer, placing even greater demands on the available memory.

EASY SCRIPT

A very large number of word processing packages have been written for the Commodore 64, but by far the most popular of these is Easy Script (mainly because Commodore bundles it with the disk drive).

The main problem for anyone wishing to write a word processing package for the Commodore 64 is the screen display, which can only produce a maximum character resolution of 40 by 25. The programmer is therefore forced to make the best of 40 columns, meaning that Easy Script doesn't have the luxury of help screens, which would otherwise occupy too much room on the screen. Furthermore, screen limitations prevent Easy


EDIT :MODE L:001 C:001
nb:adventure 51
vp2:t153
im4:rm76:jul
cn1:sp0
DIGITAYA - COMPLETE LISTING
cn0:sp1
Introduction
In this, the final week of the adve
he complete listing for Digitaya, a long
es described during the design of Haunte
Sections of the Digitaya listing ha
or Haunted Forest. The structure of Digi
unted Forest, but the game is on a much
itaya involved a map of 100 locations,
a computer, complete with memory, data
dware found inside an average home compu
the game starts the player finds himsel
ck down the mysterious Digitaya which is
he. The player must negotiate many hazar

LIZ HEANEY

Script from having any wordwrap facility, although it is possible to view the text wordwrapped prior to it being sent to the printer.

In order to overcome this problem, the programmers of Easy Script have chosen to use the screen as a 'window' on the document. The margins can be set up to 240 columns across, and when a line reaches the 40-column limit, the screen will scroll sideways until the margin limitation is reached. At this point, the screen will abruptly return to the beginning of the next line. Although this method isn't ideal, you can get used to it or, alternatively, write in 40 columns and then reformat the text before it is sent to the printer.

The various print formatting and editing commands essential to a word processing program are accessed via the Commodore 64's function keys. Pressing one will get you into a particular mode, and if followed by a keypress, will select the appropriate option. Thus editing commands are accessed by pressing F1, which is shown by Edit flashing at the top of the screen. Text and page layout are selected by pressing F3, which produces an inverse star at the cursor position. The codes to set the margins, justify the text and so on can then follow. Easy Script makes great use of these and other commands embedded in the text, which are only acted upon when the document is sent to the printer. This means that a number of the functions won't be seen on screen, reducing the ability to see what the final printed document will look like.

Easy Script contains almost all of the features you'd expect from a word processing package, such as full block manipulations, which includes copying, transferring and separate filing of the blocks, although you should note that some features work a lot better than others. The Find/Replace function, for instance, is notoriously slow, particularly when using linked files that have to be loaded separately and then searched.

TASWORD II

A word processing program that is widely used on the Amstrad range and the Spectrum+ is Tasword II. On loading, this program bears a striking resemblance to WordStar. At the top of the screen is a Help menu containing a list of the commands (and their control characters), divided into logical sections. Beneath this menu is the 16 by 80 character area set aside for writing text.

Tasword II is a powerful word processing package. The program contains a large number of commands that enable the user to format the text on the screen. Text can be moved right, left or centred on screen, and the margins set to a maximum of 128 columns. If the margins are reset, it's possible to rejustify the text to fit in with the new format. Wordwrap is implemented within Tasword, automatically rejustifying the line if a word is taken over to the next. The program is unusual in that the additional spaces are included only on the second half of the line.

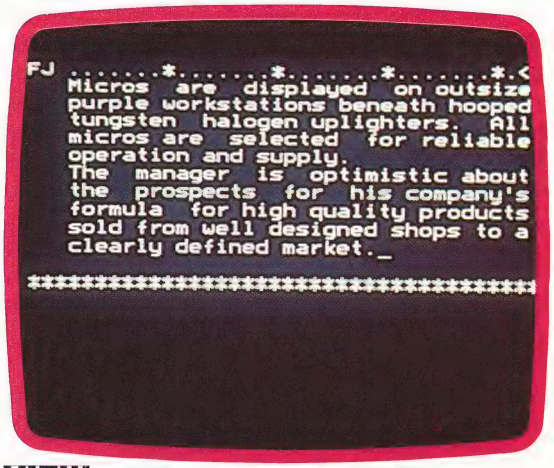
The program also facilitates block commands, although the list is not extensive. Once defined, blocks can be moved, copied or deleted, but the commands suffer from a problem that occurs in a number of features of Tasword II — the program delimits blocks in terms of lines. Thus when the user defines a block as starting midway through a line, the program will move the entire line, including the words that were meant to be outside the block area.

A similar problem is encountered when using the insert mode. Many word processors will automatically insert the text at the cursor position and move the remainder of the text to the right. Unfortunately, Tasword will either allow a single character to be inserted or provide an entire line. Furthermore, this extra line will appear under the present one, again ignoring the fact that the cursor may be in the middle of a line rather than at the end of it.

Where Tasword II really scores is in the wide number of printer control features it offers. To begin with, users are able to customise, or 'install', their own version of Tasword. This is possible by defining the page layout — setting the number of lines on a page and so on — and redefining both the print and the printer control characters.

Tasword has a wide range of printer control characters based around the Epson standard codes. They are accessed by pressing CTRL and the Space bar together, followed by the appropriate letter, with upper case turning the option on and lower case switching it off. There are 20 codes available through the program, ranging from variable line spacing to various forms of emphasised printing — such as underline, bold strike and italics.

Additional founts such as Lectura Light and Compacta are catered for by the Tasword printer control characters, but are not available under the program itself. In order to access these, the manufacturer, Tasman, has introduced a second program that can be run with Tasword called Tasprint, which can be configured for a wide range of dot matrix printers. Once configured, Tasprint can be loaded before Tasword and will assign itself an area of memory that will not be overwritten by Tasword, after which you are able to access the five additional founts. Naturally, you pay the price for these additional founts with a reduced text area; Tasprint takes up around five of the 13 Kbytes that would otherwise be available.


FJ .....*.....*.....*.....*.<
Micros are displayed on outsize
purple workstations beneath hooped
tungsten halogen uplighters. All
micros are selected for reliable
operation and supply.
The manager is optimistic about
the prospects for his company's
formula for high quality products
sold from well designed shops to a
clearly defined market._

VIEW

The BBC Micro has a somewhat different problem to overcome than that of the Commodore 64. Its screen display uses a large quantity of memory which, on the standard Model B, is in short supply. Like a number of other companies, AcornSoft has chosen to supply its word processing package, View, on ROM to fit into one of the BBC Micro's sideways sockets. This doesn't take up any of the available memory space and yet avoids the problem encountered by a number of large word processing systems, which continually have to read extra programs from disk.

This is an important consideration. Should you choose to have an 80-column display, the best that can be hoped for is just under 10,000 characters in memory. If an eight-Kbyte program was included in this, the text area would be considerably reduced, but you don't have to use 80-column mode with View. The screen can be used as a window on the document, as with Easy Script and most other good packages.

View is undoubtedly one of the better word

processing programs available on any home micro. Instead of having the help screen, you are provided with an overlay that fits over the function keys, since almost all of the word processing commands are accessed via these keys, either singly or in conjunction with the Shift or Control keys.

Among the additional features included in View is the facility allowing you to define your own 'macros'. These are short programs (created using View commands and text), which can be used to provide you with additional commands by way of a user-defined two-letter code.

The program also includes commands that perform such functions as word counting and formatting, continuous processing (which allows you to edit a document from disk or tape and then pass it directly to another file), and a full set of Find/Replace commands. These not only perform the usual functions but also permit 'wildcard' characters.

Pocket Money

It is a measure of the seriousness with which the Amstrad range of computers is now being regarded that a number of software companies which had previously only manufactured programs for business applications are now 'customising' their CP/M programs to run on the Amstrad machines. One of the greatest coups for Alan Sugar's company is that MicroPro has developed a version of WordStar, the leading professional word processing package, to run on the CPC 464 and 664 computers.

Although Pocket WordStar runs with less memory than the original package, this does not mean that it has lost many of the features of its parent program. On the contrary, almost all the facilities provided on the full WordStar package are also implemented on Pocket WordStar. The only features that are missing from the customised Amstrad version are those which require more memory than the computer has available — for example, the facility that allows the user to Print a File and Edit at the same time.

MicroPro has managed to achieve this by retaining a number of the submenus on disk, which are loaded as required. This, of course, means that Pocket WordStar is somewhat slower than its predecessor, but this is a small price to pay for an excellent word processing package on such a low-cost machine

EASY SCRIPT

Wordwrap ★

Easy Script only supports wordwrap when the document is being sent to the printer.

Block Movement ★★★★

The program allows transfer, copy and separate filing.

On-Screen Help

Not available.

80-Column Screen

Easy Script has a 40-column screen.

Word Count

Easy Script has no word count facility.

Find/Replace ★★

Up to 32 characters can be searched for, although the facility is extremely slow.

WYSIWYG ★★

Although Easy Script allows the user to examine the page layout, the low-screen resolution limits its usefulness.

Mailshot Facility ★★

The package incorporates a simple Mail Merge facility but does not permit the printing of address labels.

Spelling Checker ★★★

A sister package, known as Easy Spell, is available from the manufacurers.

Founts Available ★★★

Easy Script supports enlarged, condensed, reversed and bold print.

File Linking ★★★★★

The program supports commands to link files, either from cassette or disk, for editing or printing.

TASWORD II

Wordwrap ★★★★

Wordwrap is fully supported, with automatic justification.

Block Movement ★★★

Tasword II permits transfer, copy and deletion functions, but not separate filing.

On-Screen Help ★★★★★

On-screen help is provided by a window at the top of the screen, which can be scrolled to display all of the available commands.

80-Column Screen ★★★★★

The program displays a full 80-column screen with a maximum right margin value of 128.

Word Count ★★★★

Constantly available.

Find/Replace ★★

TasWord II allows single word search only.

WYSIWYG ★★

A number of print format options are recognised as control characters only by the printer, and these cannot be displayed on the screen.

Mailshot Facility

Not available.

Spelling Checker

Not available.

Founts Available ★★★★★

Tasword II implements several different founts, with additional styles provided on Tasprint.

File Linking ★

The program only allows a file to be added to the bottom of another.

VIEW

Wordwrap ★★★★

Full wordwrap is supported, although there is no command to turn off this function.

Block Movement ★★★★

The block operations permitted by View include deletion, move and copy.

On-Screen Help ★

View offers no on-screen help, but does provide an overlay for the function keys.

80-Column Screen ★★★

View supports a 76-column screen. However, this drastically reduces the amount of memory left for text.

Word Count ★★★★

Although View has a word count facility (which actually counts spaces) this is not a constant display.

Find/Replace ★★★★½

View can only find and replace single words, although it also supports wildcard characters.

WYSIWYG ★★★★

The program allows full display of the document as it will appear on the page, except for the 'highlight' facilities (underlining the bold print).

Mailshot Facility

Not available.

Spelling Checker

Not available, but AcornSoft have promised that one will be provided in the future.

Founts Available ★

Bold is the only additional character emphasis available.

File Linking ★★★★★

The Macro commands allow a wide range of file linking facilities.

CHANGING ADDRESS

We began our investigation of the range of 68000 addressing modes in the previous instalment (see page 1715). Having discussed the absolute, register and register indirect modes, we now turn our attention to the four remaining modes, and illustrate their differences with short pieces of code.

To begin, let's briefly review the modes we have already discussed:

- *Absolute addressing:* In which the source or destination operand is a memory address.
- *Register addressing:* The source or destination is a register.
- *Register indirect addressing (with a pointer):* We point to the source or destination object.
- *Register indirect addressing (with post-increment or pre-decrement):* We step up or down a data list.

Now let's consider *immediate* mode, in which we store a constant next to the instruction.

- *Immediate addressing:* In this mode the source operand is a constant. For example, MOVE.W #$43,D0, which instructs that the constant (specified by the # symbol) $43 ($ means hex) is moved into D0. The addressing mode is said to be immediate because the constant word is stored next to the MOVE instruction. Of course, for byte operands the extension takes one byte, and for long words four bytes.

This immediate mode is very useful for setting up a constant — say for a loop with four iterations. But note that there may be occasions when it is better to declare a constant in a named location and use absolute addressing to this location on every reference. For example:

```
MOVE    COUNT, D0
MOVE    COUNT, D5
COUNT   DC.W   4
```

This may be preferred to immediate mode addressing on each occasion if the value of COUNT is likely to change. With the absolute method we just change the contents of COUNT; otherwise we have to search through the code for all immediate mode references using, say, 4, and this can be tedious and prone to error.

With certain instructions *quick* immediate mode is permitted, where the constant is contained within the instruction in one word. Both the following examples:

```
ADDQ    #3,D0
SUBQ    #1,D3
```

would be coded into one word (and hence, executed quicker because we have the constant with the instruction). Note, however, that the constants with ADDQ and SUBQ can only be in the range #1 to #8.

The instruction MOVE also has a quick version where the range of the constant is between -128 and +127. Thus, MOVEQ #98,D4 sets D4 up to decimal 98.

To refresh your memory, here is an example that uses all the addressing modes we have looked at so far:

```
1                LEA    OUTPUT,A1    Absolute and
                                     register
2                MOVE   A1,A2        Register
3                MOVE   -(A1),D3     Pre-dec and
                                     register
4                ADDQ   #3,D3        Quick immediate
                                     and register
5                MOVE   D3,(A2)+     Register and
                                     post-inc
6                ADD    #6,D3        Immediate and
                                     register
7                MOVE   D3, (A2)     Register and
                                     indirect
8
9    INPUT       DC.W   #6           Constant 6
10   OUTPUT      DS.W   2            Space for two
                                     words
```

The INPUT location contains a constant of 6 and the OUTPUT address has a space for two words.

It is interesting to consider what is contained in OUTPUT and OUTPUT+2 once this piece of code has been executed. Lines 1 and 2 set up A1 and A2 to point to OUTPUT. Line 3 will adjust A1 to point to INPUT before addressing its operand, and so 6 will be loaded into D3. Line 4 will add 3 to D3, making it contain 9.

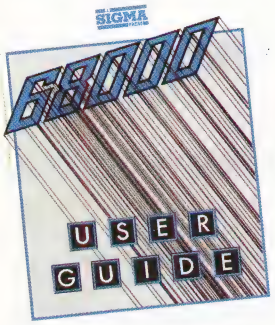
At line 5, D3 will be loaded into OUTPUT, since A2 is pointing at OUTPUT. After this has been done, A2 is incremented by two to point to the second OUTPUT word. Line 6 adds 6 to the contents of D3 (making a total value of 15), and this will be loaded into the second OUTPUT word.

- *Indirect addressing (with displacement and index):* We need first to consider what the terms 'displacement' and 'index' mean here. When dealing with the 68000, displacement refers to a fixed 'offset' or 'distance' from some base reference point, and index indicates a variable displacement via a register.

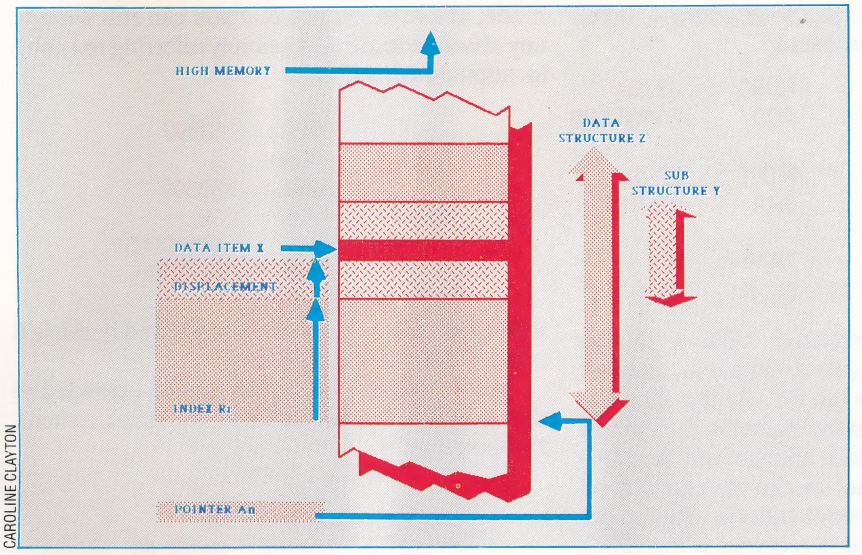
The difference between these two terms is illustrated in the diagram. This shows:

1) a structured data item called Z, which is pointed to by address register An;
2) an internal substructure called Y, which is indexed via an index register called Ri (an address or data register);
3) and an element of Y called X, which can be accessed with a fixed displacement.

Examples of these data structures are:



Finding Your Way
The 68000 User Guide contains brief details of the 68000 instructions and modes, but is more useful for its suggestions on how the 68000's facilities are best employed in a number of sample applications. Published by Sigma Press; ISBN 1 85058 001 4; price £8.95



CAROLINE CLAYTON

1) Z: an array of records, or a two-dimensional array;
2) Y: the record itself, or a row of an array;
3) X: the record element, or the array integer.

These are structured data items that are common in modern high-level languages such as PASCAL and ADA, and hence the ability to reference such items easily is very important. Of course, there is no reason why the assembler program should not structure data in a similar way — we certainly have the means to do so with the 68000!

Let's return to our structured data example once more. You may have noted that the address of X is formed by the sum of An+Ri+displacement, and we can alter An and Ri as the program executes. We can now look at some more concrete addressing examples, starting with one that illustrates the simplest mode — indirect addressing with displacement:

```
MOVE.W DISP(A0),D1
```

where previously DISP would have been defined as a symbolic name of, say, value 6. This instruction uses indirect addressing with a displacement of 6 as the source addressing mode. If, for example, A0 was to point to address $1000 (1000 hex), then D1 will be loaded with the contents of address $1006. Note that because a whole word extension is allowed with the instruction for the displacement, we can have a full 16-bit signed displacement integer (a value in the range +32767 to −32768).

Now let's look at the most powerful addressing mode available to the 68000 programmer — displacement — as in the following example:

```
MOVE.W  DISP(A0,D0.W),D1
```

In this case, the source address is formed by adding together the 'base' register, A0, with the index register, D0, and the fixed displacement, DISP. Note, however, in this case that DISP can only be a signed eight-bit integer, although the index register can be a full 32-bit long word.

Notice, when you compare these two addressing modes (indirect and displacement, with and without index), that the displacement is always fixed at assembly time. However, if you require to alter a displacement dynamically, then you always have the option of using:

```
MOVE.W  0(A0,D1),D2
```

where D1 now effectively becomes the displacement (which we can alter in the program execution), and the fixed displacement is zero.

Finally, it should be noted that the 68000 instruction set offers these two powerful addressing modes for most of the more common instructions. So, let's look at an addressing example using displacements and indexes, and with bytes as the data attributes.

```
1        LEA     LIST,A1
2        MOVEQ   #4,D1           D1:=4
3        MOVE.B  2(A1),D6        D6:=3
4        ADD.B   0(A1,D1),D6     D6:=8
5        MOVE.B  D6,4(A1,D1)     LIST+8:=8
6
7  LIST  DC.B    1,2,3,4,5,6,7,8,0
```

A1 is set to point to LIST on line 1 and then D1 is set to 4. On line 3, a displacement of 2 is added to the pointer in A1, so the third element of LIST is loaded into D6. Then, with an index of 4 in D1, the contents of element 5 are added to D6, which is then copied into LIST+8.

Notice that on line 4 we have used a displacement of 0 — this gives us the ability to use D1 as an index register, which we could change when the program runs — if we wanted to do this.

- *PC relative addressing:* Before moving on to consider PC relative modes of addressing in detail we need to look at some of the assembler directives. These are instructions to the assembler that do not directly produce executable code, but which influence such factors as the listing format or the definition of symbols. One such directive is specifically concerned with the program start address and the type of code produced. This is the ORG, or origin, assembler directive.

Normally, we would specify the start address using, for example:

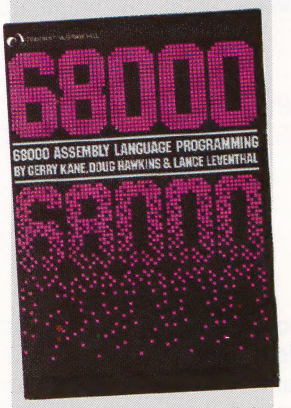
```
ORG   $1000
```

This sets the start address (or really the load address for the binary loader) to $1000 and all the subsequent code is loaded in by sequentially increasing addresses. If another ORG directive is encountered, then this would, of course, set a new load address at that point. A variation on this would be:

```
ORG.L   $2000
```

where all forward references would be assumed to be full long words, and hence any such absolute references would occupy two whole words.

The RORG directive defines a load address from which all memory references are taken to be relative to the program counter, rather than being absolute. The principle of this type of code is that no matter where the code is loaded, the memory



Comprehensive Tutorial
68000 Assembly Language Programming is a complete guide to the subject published by Osborne/McGraw- Hill. Experienced programmers will find the sections on assembly programming theory largely unnecessary and the style of the book is uncompromisingly tutorial. But what the book lacks in informality it makes up for by including all the information you are likely to need. One good feature is the use of bold-type paragraph headings, which enable you to skip rapidly through sections searching for a particular passage. ISBN 0-931988-62-4; price £19.95

references are relative to that load address. Take, for example, the following listing:

```
                        RORG    $2000
2000 D47A    START      ADD     CONST1,D2
2002 000A
2004 3202               MOVE    D2,D1
2006 6000               BRA     START
2008 FFFB
200A 4E40               TRAP    #0
200C 0026    CONST1     DC.W    38
```

Here, CONST1 is referenced with a 16-bit displacement of A hex (10 decimal) in location 2002, which is added to the PC. At the time the displacement is loaded in the example, the PC will be 2002 so the source data address will be 200C. Notice that there are no special instructions or differences in the addressing syntax in order to get PC relative code; all that is required is the RORG assembler directive.

There are a few other points about this code that should be noted. First of all the BRA (branch always) instruction has a displacement associated with it in address 2008, which, when added to the PC, will give the operand address of START. This means that the BRA instruction will always give PC relative code. The second thing to note is the TRAP instruction. This is used, in this context, as a halt instruction, with the bonus that entry is made to a monitor that prints out registers and enables the user to examine memory. This instruction, if your monitor allows it, can be very useful.

Let's look at another example, in which the PC relative mode addressing gives a negative displacement:

```
                        RORG    $2000
2000 0026    CONST1     DC.W    38
2002 DA7A    START      ADD     CONST1,D2
2004 FFFC
2006 3202               MOVE    D2,D1
2008 6000               BRA     START
200A FFF8
200C 4E40               TRAP    #0
```

Here, the reference to CONST1 is a negative displacement in address $2004. (You will recognise, of course, that FFFC is negative because the sign is set. The magnitude — or size — of the number can be found by inverting and adding one; therefore, FFFC is −6 decimal.)

One restriction on this addressing is that it is only allowed as a source operand. For example:

```
            RORG    $2000
TOTAL       DC.W    0
START       ADD     D2,TOTAL
```

would generate an assembler error. This means that this type of addressing is very suitable for code which has to be put in a ROM (since we cannot write to a ROM) at a fixed address, but yet can be tested at any suitable RAM location (where, of course, we can read and write).

You may think that it is unnecessarily restrictive to be unable to write to locations in this addressing mode. However, notice that you can still write to any absolute location by simply referring to such a location. For example:

```
            ORG     $1000
TOTAL       DS.W    0
            RORG    $2000
CONST1      DC.W    38
START       ADD     CONST1,D2
            MOVE    D2,TOTAL
```

where the reference to TOTAL is allowed because it is in an absolute address area.

The PC relative mode can also be extended so that an index can be used with the displacement. For example:

```
            RORG    $3000
INDEX       DC.W    10
START       MOVEA   INDEX,A5
            ADD     6(A5),D2
```

In this case we have had to use the instruction MOVEA, which loads A5 with the contents of memory location INDEX. We could not use LEA as this would take the address of INDEX (not allowed anyway in PC relative mode). Of course, a straight MOVE instruction would be illegal, as we have already seen (an address register is not allowed as a destination operand).

So let's go back now to the example where the source operand for the ADD instruction will be the PC value after the ADD instruction plus the index register (A5 here) with a displacement of 6. In this example, the source address will be 16 bytes after the address containing the extension word 6.

One restriction on this mode, however, is that the displacement is formed in eight bits within the opcode, so that this restricts our displacement to +127 or −128 bytes.

The significance of this PC relative mode is that the code can be executed anywhere in memory, as we have seen in the case of writing code for the ROM. Notice, however, that this form of addressing is extremely useful more generally for writing position-independent code. This form of code may be necessary for large multi-module programs where the position of a module in memory is not fixed until that module is loaded into memory. Of course, for large multi-programming systems, we may need a memory management unit with extra facilities (Motorola provides this), but for any simpler memory scheme the PC relative mode is very important.

- *Implied addressing:* This mode of addressing shouldn't cause any trouble because the opcode specifies the registers to be used. For example, the RTS instruction will affect the PC and stack pointer; BRA affects the PC.

In the next instalment, we will be looking at the 68000 instruction set, starting with the data-copying and arithmetic instructions. Further articles will deal with logical instructions and program control to give you a thorough appreciation of programming the 68000.

Motorola 68000 Instruction Set

Here, courtesy of Motorola Inc, we present a summary of the 68000's instruction set

Instruction Set Summary

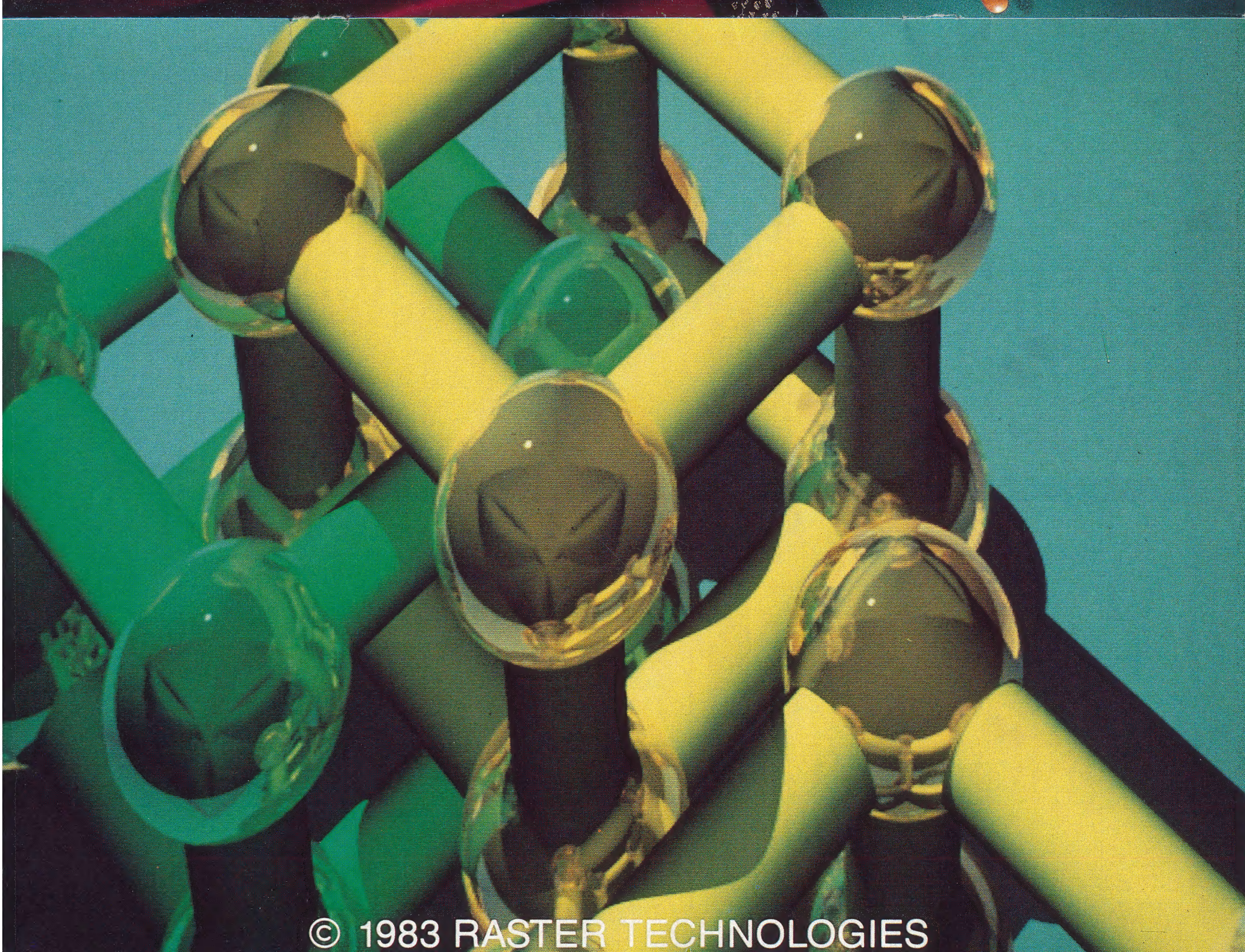
Mnemonic	Description
ABCD	Add Decimal With Extend
ADD	Add
AND	Logical And
ASL	Arithmetic Shift Left
ASR	Arithmetic Shift Right
Bcc	Branch Conditionally
BCHG	Bit Test and Change
BCLR	Bit Test and Clear
BRA	Branch Always
BSET	Bit Test and Set
BSR	Branch to Subroutine
BTST	Bit Test
CHK	Check Register Against Bounds
CLR	Clear Operand
CMP	Compare
DBcc	Test Condition, Decrement and Branch
DIVS	Signed Divide
DIVU	Unsigned Divide
EOR	Exclusive Or
EXG	Exchange Registers
EXT	Sign Extend
JMP	Jump
JSR	Jump to Subroutine
LEA	Load Effective Address
LINK	Link Stack
LSL	Logical Shift Left
LSR	Logical Shift Right

Mnemonic	Description
MOVE	Move
MULS	Signed Multiply
MULU	Unsigned Multiply
NBCD	Negate Decimal with Extend
NEG	Negate
NOP	No Operation
NOT	One's Complement
OR	Logical Or
PEA	Push Effective Address
RESET	Reset External Devices
ROL	Rotate Left without Extend
ROR	Rotate Right without Extend
ROXL	Rotate Left with Extend
ROXR	Rotate Right with Extend
RTE	Return from Exception
RTR	Return and Restore
RTS	Return from Subroutine
SBCD	Subtract Decimal with Extend
Scc	Set Conditional
STOP	Stop
SUB	Subtract
SWAP	Swap Data Register Halves
TAS	Test and Set Operand
TRAP	Trap
TRAPV	Trap on Overflow
TST	Test
UNLK	Unlink

Variations Of Instruction Types

Instruction Type	Variation	Description
ADD	**ADD**	Add
	ADDA	Add Address
	ADDQ	Add Quick
	ADDI	Add Immediate
	ADDX	Add with Extend
AND	**AND**	Logical And
	ANDI	And Immediate
	ANDI to CCR	And Immediate to Condition Codes
	ANDI to SR	And Immediate to Status Register
CMP	**CMP**	Compare
	CMPA	Compare Address
	CMPM	Compare Memory
	CMPI	Compare Immediate
EOR	**EOR**	Exclusive Or
	EORI	Exclusive Or Immediate
	EORI to CCR	Exclusive Or Immediate to Condition Codes
	EORI to SR	Exclusive Or Immediate to Status Register

Instruction Type	Variation	Description
MOVE	**MOVE**	Move
	MOVEA	Move Address
	MOVEM	Move Multiple Registers
	MOVEP	Move Peripheral Data
	MOVEQ	Move Quick
	MOVE from SR	Move from Status Register
	MOVE to SR	Move to Status Register
	MOVE to CCR	Move to Condition Codes
	MOVE USP	Move User Stack Pointer
NEG	**NEG**	Negate
	NEGX	Negate with Extend
OR	**OR**	Logical Or
	ORI	Or Immediate
	ORI to CCR	Or Immediate to Condition Codes
	ORI to SR	Or Immediate to Status Register
SUB	**SUB**	Subtract
	SUBA	Subtract Address
	SUBI	Subtract Immediate
	SUBQ	Subtract Quick
	SUBX	Subtract with Extend


© 1983 RASTER TECHNOLOGIES